NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A PROTOCOL FOR BUILDING A NETWORK ACCESS CONTROLLER (NAC) FOR "IP OVER ATM"

by

Ioannis Kondoulis

September 1998

Thesis Advisor:	Geoffrey Xie
Second Reader:	Cynthia Irvine

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE **A PROTOCOL FOR BUILDING A NETWORK ACCESS CONTROLLER (NAC) FOR "IP OVER ATM"**	5. FUNDING NUMBERS
6. AUTHOR(S) Lt Ioannis Kondoulis, Hellenic Navy	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The implementation of *label swapping* packet-forwarding technology increases the vulnerability to insider attacks. These attacks refer to unauthorized access from within an enclave to the outside network. In this thesis we propose a protocol to counter this category of attacks. The proposed protocol provides a means for fast packet authentication. High speed is achieved by the use of a *trailer*, which allows packet filtering at Layer 2, and the use of cheap and fast message digest algorithms. To overcome the weaknesses of a 128-bit message digest algorithm, each key is designed to have a very short cryptoperiod. Such fast rekeying is implemeted by key caching (the host has a table of keys). Initial performance measurements indicated that it is possible to use our protocol while maintaining very high data throughput. Specifically, our protocol implements an authentication module, called Network Access Controller (NAC). The NAC's modular nature allows it to be easily integrated with a variety of routing technologies and other security mechanisms while remaining totally independent of them.

14. SUBJECT TERMS protocol, Network Access Controller (NAC), Internet Protocol (IP) ,Asynchronous Transfer Mode (ATM)	15. NUMBER OF PAGES 231
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFI- CATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

Approved for public release; distribution is unlimited

A PROTOCOL FOR BUILDING A NETWORK ACCESS CONTROLLER (NAC) FOR "IP OVER ATM"

Ioannis Kondoulis
Lieutenant, Hellenic Navy

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1998

Author: ______________________
Ioannis Kondoulis

Approved by: G Xie
Geoffrey Xie, Thesis Advisor

Cynthia E. Irvine
Cynthia Irvine, Second reader

Dan C Boger
Dan Boger, Chair
Department of Computer Science

ABSTRACT

The implementation of *label swapping* packet-forwarding technology increases the vulnerability to insider attacks. These attacks refer to unauthorized access from within an enclave to the outside network. In this thesis we propose a protocol to counter this category of attacks. The proposed protocol provides a means for fast packet authentication. High speed is achieved by the use of a *trailer*, which allows packet filtering at Layer 2, and the use of cheap and fast message digest algorithms. To overcome the weaknesses of a 128-bit message digest algorithm, each key is designed to have a very short cryptoperiod. Such fast rekeying is implemeted by key caching (the host has a table of keys). Initial performance measurements indicated that it is possible to use our protocol while maintaining very high data throughput. Specifically, our protocol implements an authentication module, called Network Access Controller (NAC). The NAC's modular nature allows it to be easily integrated with a variety of routing technologies and other security mechanisms while remaining totally independent of them.

DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at risk of the user.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

This thesis explores how the implementation of "Tag Switching" – a technology described in Internet RFC 2105 (Rekhter, 1997) – for Internet Protocol (IP) over Asynchronous Transfer Mode (ATM) networks can increase the risk of an *insider-attack*. We present a protocol that counters this kind of attacks using a security "gateway" termed *Network Access Controller* (NAC). We also present a prototype simulation of the protocol in C++ for demonstration purposes. Further, we evaluated the performance (in terms of overhead) of the protocol.

A. BACKROUND

1. Networking

a. Packet forwarding and "Tag Switching"

The explosive growth of the Internet has created the demand for higher bandwidth and, consequently, for faster and more effective routing and packet forwarding techniques. Specifically, today's routers are processor-based thus, computation takes place at them. In order to decide how to forward a packet they use the Network Layer (Layer-3 of the OSI model) information contained in a packet's header and in conjunction with a routing table. A good analogy of such routing in real life would be a company transporting goods (messages) by trucks (packets). Each truck has a final destination IP address and at each crossroad (router) its driver reports its destination (the packet header being parsed) and asks a traffic coordinator (routing process) for directions. The traffic coordinator has a table of destinations and the corresponding directions for leaving the crossroad (routing table). He looks up the table and directs the driver to the right road (routing decision).

The routing table at a router must be updated periodically to reflect topological changes or road fluctuations in the network. Each packet's header has to be parsed by the routing process. All these functions are currently implemented by specialized software. In summary, Layer-3 processing is complex, computationally costly (time consuming) and the equipment required is expensive. Nevertheless, it has been the choice because it is very flexible and very functional as shown by Internet's exponential growth.

Switching, on the other hand, is faster and less complex than routing, and the equipment (switches) are cheaper. It has been reported that for the same throughput, routing costs 20 times as much as switching (Newman, 1998). Switches use the Data-Link Layer (Layer-2 of the OSI model) information contained in a packet's header in order to forward the packet to its destination according to fixed rules. This functionality is implemented, usually, in hardware. An analogy of switching with the real world can be found again using the transportation company paradigm. A racecar leaves the company's headquarters and finds the best way to the destination (signalling). The racecar carries a paintbrush that paints the road with a fluorescent color. The truck drivers now need not know their destination; they only have to follow the paint. The only problem with Layer-2 switching is that it is not as flexible and functional as Network Layer routing because any change in the route would require repainting the entire road from the beginning.

Recently a number of new technologies have emerged that combine the Layer-2 switching with the Layer-3 routing for performance, flexibility and functionality. They include Cisco's *Tag Switching* (Rekhter, 1997), Ipsilon's *IP switching* (Newman, 1998) and 3Com's *Fast IP* (Seaman, 1997). Although there is not yet a standard, all of these technologies follow the so-called *label-swapping* paradigm for packet forwarding. This paradigm can be described simply as follows: A *label* or "tag" (i.e., an integer) is assigned to a packet[1] according to the *flow*[2] to which it belongs. Packet forwarding is based solely on this label. The implementation of Tag Switching and Fast IP is independent of the lower network layers[3] (Layers 1 and 2 of the OSI model). Therefore, they are flexible and adaptable.

[1] The label is carried by the packet as part of its header.

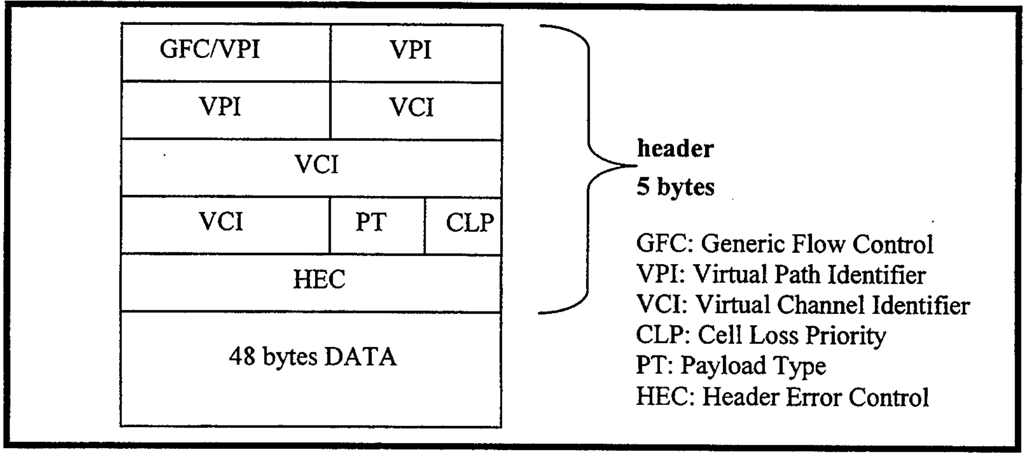
[2] A sequence of packets with common attributes e.g., those carrying data for a video stream or a multimedia file.

[3] Only *IP switching* is closely coupled with ATM.

b. Asynchronous Transfer Mode

The Asynchronous Transfer Mode (ATM) protocol is a connection-oriented protocol for computer communication, analogous to the telephony. It is designed to carry data at extremely high speeds (e.g., 622Mb/s). ATM uses fixed size (53 byte) packets called *cells* (Figure 1), which travel in virtual channel connections (VCCs), established between senders and receivers. In ATM, the cells that travel in one VCC constitute a *flow*. The setup and teardown of a VCC are performed by *signalling* functions. The term *virtual* is used because a single physical medium (e.g., fiber optic) can be utilized to accommodate the traffic of many Virtual Channels (VCs). Once a VCC has been established, the switching of the cells inside an ATM switch is done in hardware.

The *Virtual Channel Identifier* (VCI), an integer stored in each cell's four-byte header is used by switches to forward cells to the right VCs. Each cell's incoming VCI will be translated into a new one (the outgoing VCI) inside each switch. Each switch maintains a table that maps the incoming VCIs to entries containing two elements: a new VCI and the appropriate output port. These tables are updated during the *connection setup*[1] phase. It is obvious that ATM switching follows the *label swapping* forwarding paradigm.



GFC: Generic Flow Control
VPI: Virtual Path Identifier
VCI: Virtual Channel Identifier
CLP: Cell Loss Priority
PT: Payload Type
HEC: Header Error Control

Figure 1. ATM cell format

[1] A *signalling* function. Signalling functions are responsible for setting up, maintaining and tearing down a connection and use *permanent* or *semi-permanent* VCs dedicated to signalling.

Another attractive feature of ATM is that the users can request quality of service (QoS) for a VCC. Four *ATM service classes* provide different levels of QoS. They are:

- *Constant Bit Rate* (CBR): guaranteed transmission rate for applications that need it, e.g., 12 Mb/s for a videoconferencing VC.
- *Variable Bit Rate* (VBR) (*real-time* and *non-real-time* versions): a *standard* transmission rate (SCR) is assigned to a connection. This rate can momentarily rise to the *peak* transmission rate (PCR) in order to accommodate the needs in bandwidth of bursty transmissions. Real time (RT) version is required when an application needs to maintain a timing relationship between two users e.g., in voice applications. Non-Real Time (NRT) can be used when information retrieval at a later time will not cause any problem e.g., store and forward video.
- Available Bit Rate (ABR): low *cell loss*[1] but the available bit rate fluctuates, depending on the network resource availability.
- Unspecified Bit Rate (UBR): these cells are the first ones lost when congestion occurs.

Although ATM is being heavily promoted, especially by the telecommunications community, its deployment has been slow mainly because of the existence and success of the IP. It is clear today that the ATM technology will not totally replace IP. However, the advantages of ATM – high speed and QoS – led to the development of techniques that combine IP and ATM. These are the Classical IP over ATM described in RFC 1577 (Laubach, 1994) and the LAN emulation (LANE)(AF-LANE 0084.000, 1997).

[1] Cell loss can happen when *congestion* occurs, e.g., when cells arrive at a switch at higher rates than the ones that the switch can forward them, this results in *buffer overflow* and consequently in cell loss.

2. Network Security

The Internet is often called "the network of networks". Interconnectivity at this scale has created the need to protect an organization's network against malicious or accidental behaviors. The same argument holds for a subnet connected to a corporate intranet and for a site connected to a subnet. In order to safeguard a network or a network segment, a security policy must be set for the network. After setting the policy, the administrator(s) must choose tools that help enforce the security policy. These tools include boundary controllers, encryption (and hashing) algorithms, access control and authentication mechanisms.

A firewall isolates an internal network segment from the outside world and allows access only to authorized traffic (Figure 2). Additionally, it controls access from the internal network to the Internet (or an intranet).

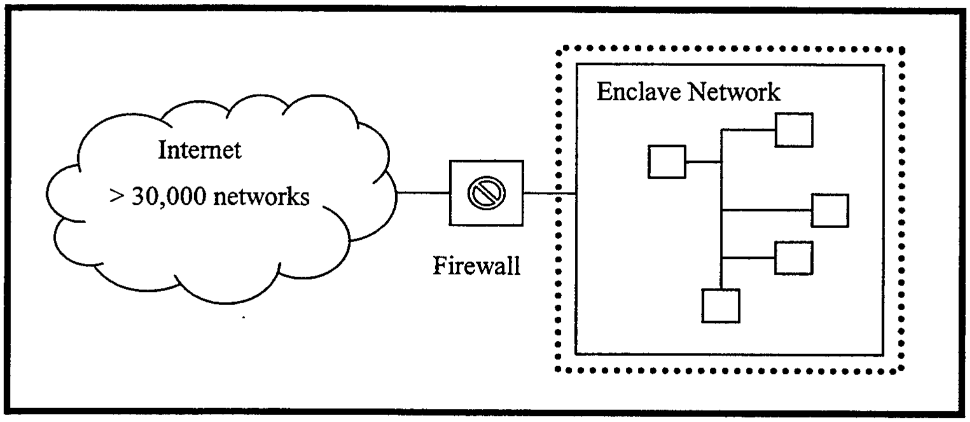


Figure 2. A firewall protecting an enclave. After (Hare, 1996).

Below are two examples from (Hare, 1996) that demonstrate this functionality. They are implemented in the freely available firewall program *TCP Wrapper*.

- Example 1: the entry in the*/etc/hosts.allow* file:

  fingerd, telnetd: 144.19.74.1, 144.20

  allows host 144.19.74.1 and all hosts on class B network 144.20.0.0 to access *telnet* and *finger* services.

- Example 2: the entry in the */etc/hosts.deny* file:

```
Fingerd, telnetd : .hacker.org
```

denies access to *telnet* and *finger* services to all hosts in the domain HACKER.ORG.

Encryption algorithms can be used as parts of protocols that can support *confidentiality*, *integrity*, *authentication*, *non-repudiation* of information travelling between computers. There are many of them. Two widely used ones (Schneier, 1996) are:

- DES (Data Encryption Standard) is the most popular. It uses the same key for encryption and decryption (*symmetric* algorithm). The common key is *secret.* The algorithm transforms data into an unmeaningful series of bits. If the key becomes available to an attacker then the algorithm is compromised (broken).

- RSA (named after its creators – Rivest, Shamir and Adleman) is the most popular *public key* algorithm. The only *secret* key is the private key, which is known only to its owner. The *public* key is available to others. Data encrypted by a user's private key can be decrypted only by the same user's public key and data encrypted by a user's public key can be decrypted only by the same user's private key.

Message digest[1] algorithms can be used to provide message authentication and data *integrity.* The whole message is *mapped* by the keyed one-way hashing function to a (usually) small *hash value* or *message digest* or *message authentication code (MAC).* This value serves the role of the *signature* for the particular message. The most widely known message digest algorithms are the ones of the MD family (MD4 (Rivest, 1990), MD5 (Rivest, 1992)) written by Ron Rivest (RSA) and the Secure Hash Algorithm (SHA). The SHA is part of Digital Signature Standard (DSS)(NIST, 1994) which is a protocol used for creating *digital signatures.*

[1] Known also as *keyed one-way hash functions.*

Finally, the *authentication and key-exchange mechanisms,* usually implemented by centralized authentication servers, allow access to resources only to authorized users. Two popular authentication services are:

- *Kerberos* which uses conventional encryption and
- *X.509-based key exchange*, which is part of the Privacy Enhanced Mail (PEM) protocol and uses public key encryption (RSA is recommended as its underlying public key encryption algorithm).

A more detailed discussion around Network Security follows in Chapter II of this thesis.

B. SCOPE OF THE THESIS

This thesis examines the vulnerabilities introduced to a network by the implementation of Tag Switching. In particular we have focused on security problems related to unauthorized access from the internal network (*enclave*) to the outside world. We call them *insider attacks.*

Specifically we propose a protocol that will guard against this category of attacks and we evaluate the performance of the protocol. Further, a prototype simulation of the protocol in C++ is presented.

C. ORGANIZATION

The thesis is organized as follows: Chapter II gives an overview of *network security* issues related to firewalls and message authentication. Chapter III *discusses high speed networking* issues, in particular IP over ATM and Tag Switching. Chapter IV presents *insider attacks* (the problem) and our *protocol* (the proposed solution), including our assumptions. Chapter V presents the performance evaluation of the protocol. Chapter VI presents the conclusions we reached and describes future work that could follow this thesis.

Note that in Chapters II and III, we focus on the aspects of network security and high-speed networking that are related to the thesis. Both chapters are presented for background information. The simulation source code and the software engineering notes about the code are included as Appendixes. We have also included as an Appendix the measuring code that was used for the performance evaluation of the protocol.

D. MAJOR CONTRIBUTIONS OF THIS THESIS

There are three major contributions of this thesis.

- It examines the vulnerabilities of newly proposed packet forwarding technologies that use the *label-swapping paradigm* – in particular Tag Switching.
- It proposes a flexible[1] solution (protocol) that can be implemented in a straightforward manner.
- It provides an initial evaluation of how the implementation of the protocol affects the performance of Tag Switching when it is used for IP over ATM.

[1] Can be adapted to any label swapping technology.

II. NETWORK SECURITY

A. SCREENING ROUTERS AND FIREWALLS

A fundamental measure for home security is locking the door. However, one must be able to examine each person that knocks on the door and allow him/her to enter when it is desired. The same action should take place at the perimeter of a network segment (site, enclave or intranet). In the network case, arriving packets are checked (and not persons). These packets may arrive from the rest of the network, intranet and finally from the Internet. Additionally we want to control the access from inside the enclave to the outside world. Such functionality is known as *access control*, and it is implemented by devices generally known as *firewalls*. The use of firewalls is a common practice today.

A Discretionary Control Mechanism (DAC) may be in force as part of the security policy of an enclave. In the context of such a mechanism, the security officer of the enclave may decide to prohibit a user access to the outside world. For example, the ISSO may have granted Bob access privilege to the outside world, but has not done so with Alice. In this case, the packets that Alice is sending must be *denied access* to the external network.

A definition of a *firewall* is “a collection of components that are placed between two networks that create controlled isolation” (Cheswick, 1994). It may be inappropriate to use the term *firewall* for a variety of products and architectures. Nonetheless, it is now a general term in the literature. “Firewall” also has become a synonym for *Internet Firewall*, because it has been common for firewalls to be used as isolating devices between corporate or other types of networks and the Internet. In fact, firewalls can exist between any two networks.

The existence of a firewall increases the security of a network because by using a firewall one can enforce aspects of the network’s security policy. When and how access to a resource (host, network or service) is allowed depends, and directly reflects, this security policy. Detailed information on network security policy can be found in chapter 3 of (Hare, 1996). We must emphasize here that firewalls control the access to resources in *both* directions: from the outside world into the network and from the network to the outside world.

The following three main techniques are used in firewalls[1]:

- *Packet filtering* occurs when the firewall decides to forward or to block a packet based on some rules. The packet header is parsed and according to the type of the protocol and/or the values in the fields of the packet header e.g., the source or destination address, the packet is allowed to proceed or not. The firewall therefore can control the type of network traffic and the type of services available in a network segment. The term *screening routers* or *packet filter routers* are used for the devices[2] that perform this screening task. Since each set of fields in the packet header is added to the packet by a layer of the OSI Reference Model, we can identify exactly at which level the filtering functions takes place. Usually screening routers perform their screening functions at the network (a.k.a. Layer 3) and the transport layers (a.k.a. Layer 4) of the OSI Reference Model. Sometimes, however, filtering can happen at the data link (a.k.a. Layer 2) or even at the physical (a.k.a. Layer 1) layers. (Hare, 1996)

- *Circuit gateways* are used to establish, for authorized users, a channel of communication between two network segments (networks, subnets, sites). The main concern is *who* is using an application and not what information is passed by the network application. Circuit gateways are similar to packet filters. Additionally they use the services of an out-of-bound authentication scheme that provides the information needed for authenticating a user.(Atkins, 1997)

- *Application proxies* are used when it is important to control the content of the data to an application. An example of such a proxy is one used to limit FTP

[1] Details for all of the techniques can be found in (Atkins, 1997).

[2] They are commercial routers that are capable of screening packets.

users so that they can only get files from the Internet, but never push local files to the Internet. The problem with application proxies is that they are application specific and therefore, difficult to write and maintain. Application proxies can be combined together with circuit gateways in firewall products. (Atkins, 1997)

We are interested in combining the packet filtering and circuit gateways techniques in our design of the NAC.

B. MESSAGE AUTHENTICATION FUNCTIONS

Computer networks are vulnerable to a number of attacks. We describe some of these attacks in order to give a perspective of the potential threats to a network. Message authentication is one of the effective measures against a broad spectrum of network attacks. We describe three classes of message authentication in this section.

1. Network Attacks

Stallings describes the possible attacks in computer networks (Stallings, 1995):

- *Disclosure*: the contents of a message become available to an unauthorized to know person or process

- *Traffic analysis*: a pattern of traffic between the communicating entities is identified. Information such as frequency and duration of connections[1], number and size of messages can be deduced.

- *Masquerade*: an malicious source creates and inserts messages into the network. These messages may be purported to come from an authorized entity, they may be faulty acknowledgements of messages etc.

- *Content modification*: the contents of a message are changed, deleted or modified.

[1] In connection oriented communications such as ATM.

- *Sequence modification*: the sequence of messages is altered. New messages may be inserted, messages can be deleted or simply reordered.
- *Timing modification*: Messages may be delayed or replayed. An individual message (e.g., datagram), an entire session or a sequence of messages belonging to a previous, valid session can be replayed or delayed.
- *Repudiation*: a source denies that it ever sent a message or a destination denies that it ever received a message.

2. Authentication Functions

Message authentication can counter masquerade, content modification, sequence modification and timing modification. Repudiation is specifically countered *by digital signatures*. (Stallings, 1995)

Message authentication mechanisms consist of two parts:

- *Authentication function*: produces an *authenticator* (a value that authenticates the message) known also as *Message Authentication Code* (MAC).
- *Authentication protocol*: the sequence of actions that verifies the authenticity of the message. (Stallings, 1995)

The first part of a message authentication mechanism, the authentication function, can be further classified in the following three classes:

a. Message encryption

The ciphertext of the whole message is used as its authenticator. There are two different ways to authenticate a message: using a conventional (symmetric) encryption method and using a public-key encryption method.

When a conventional encryption scheme is going to be used, if a message M has to be exchanged between two parties S (source) and D (destination) and the shared secret key is K, then the following scenario takes place:

- S encrypts M using K and produces the encrypted message: $E_K(M)$.

- S sends E_K (M) to D.
- D decrypts E_K (M) using K and gets message M: D_K (E_K (M)) = M.

If the key is known only to S and D, then D can be confident that message M came from S. However, the degree of authentication is limited. The key K could have been compromised or the encrypted message could have been a *replay*. Additionally when D is an automated system, and not an intelligent person, and message M is a stream of bits, it is very difficult for D to judge for M if it is an acceptable stream of bits or not. A video or an audio stream falls into this category. Any alteration could not be detected by an automated system. The result of this weakness of automated systems in the previous example would be that D, falsely, may assume that any E_K (M) has come from S. Therefore, additional formatting of the method and additional redundancy is needed for using it in automated systems. (Stallings, 1995)

In a public-key encryption scheme if the message M is encrypted with the public key of D, denoted KpuD, no authentication can be provided. The obvious reason is that any one that knows KPuD (since it is public, and therefore, available to every one) could have send the message E_{KPuD} (M) to D. Authentication can only be provided when the sender S uses his private key, denoted KPrS, to encrypt M. The result is E_{KPrS} (M) and it can be decrypted by anyone that has S's public key KPuS. This may affect the confidentiality of message M but D now can be certain that S is the sender of M.

We can eliminate the disadvantages and get only the benefits from the two public-key encryption scenarios that we mentioned if we use – paying the additional cost in performance and complexity – the following, *double* public-key encryption, scenario:

- S encrypts M using KPrS and produces the encrypted message:

  E_{KPrS}(M) .

- S encrypts E_{KPrS} (M) using KPuD and produces the encrypted message: E_{KPuD} (E_{KPrS} (M)).
- S sends E_{KPuD}(E_{KPrS}(M)) to D.

- D decrypts E_{KPuD} (E_{KPrS} (M)) using KPrD and gets message E_{KPrS} (M): D_{KPrD}(E_{KPuD}(E_{KPrS}(M))) = E_{KPrS} (M). (this step provides *confidentiality*).
- D decrypts E_{KPrS} (M) using KPuS and gets message M: D_{KPuS} (E_{KPrS} (M)) = M. (this step provides *authentication*).

It is obvious that similar problems exist here, as with the shared key encryption scheme. Again, it is difficult for an automated D to reject a series of bits M as unmeaningful and replays can take place as well as it was possible in the conventional encryption scheme. Therefore, some refinement of the method is needed here, as well as in the conventional encryption scheme, in order to use it as an authentication scheme.

b. Cryptographic checksum

The sender S (source) and receiver D (destination) must share a secret key K (similarly to a conventional encryption scheme). S applies an *encryption-like* algorithm to the message M using the common key K. This algorithm produces a small, fixed length set of bits known as a *cryptographic checksum* or *message authentication code* (MAC). The MAC is then appended to the message M and both are sent to D. D recalculates the MAC (applying the same algorithm) for the received message, using its copy of the key K. The message is authenticated if the received MAC and the calculated MAC are the same. D can also be certain that M has not been modified (otherwise, the calculated MAC would not be the same with the MAC received). The obvious vulnerability of the method is when the secret key K has been compromised.

The method does not provide confidentiality. In the case that the communicating parties need confidentiality, as well as authentication, the message should be encrypted (it is recommended that MAC is calculated for the plain text M and not after the encryption (Stallings, 1995)).

The difference between the algorithm that is used for the MAC calculation and that of a conventional encryption function is that the former does not need to be reversible (it is a one-way encryption function) as it must be for decryption. This makes it harder to break than encryption. Additionally the result is a compact representation of the message. (Stallings, 1995)

(Davies, 1989) suggests the use of MAC in the following three cases:

- Authenticating control messages in a network. Each control messages is broadcast with its MAC. A network node, playing the role of the *authentication monitor,* that has the secret key, authenticates the message or alerts all the recipients that the message is fake.
- When a heavy load of incoming messages floods the receiver D. We cannot afford the resources to decrypt all of them. An authentication scheme that performs selective MAC checking on a random basis can be used.
- For executable programs, an attached MAC could be used for assuring their integrity.

(Stallings, 1995) adds three more rationales to the above:

- When secrecy is not required but it is still important for messages to be authenticated. An example is the Simple Network Management Protocol version 2 (SNMPv2). In SNMPv2, the functions of confidentiality and authentication are separated. The incoming messages may not need to be secret, nevertheless, they have to be authenticated, especially when they change parameters at the managed system.
- Separation of confidentiality and authentication functions allows for flexibility in the architecture. The designer may choose in which layer of the OSI model to apply the one or the other. A possible decision could be to apply authentication (MAC) at the application layer (Layer 7) and confidentiality (using encryption) at the transport layer (Layer 4).
- When we want to protect the integrity of the contents of a message after it is decrypted. A MAC for that message can ensure that its contents have not been fraudulently modified.

c. *One-way Hash Functions*

This is a variation of the previous method. A public function, *H*, is used to map each message *M*, regardless of its length, into a *hash value* or *hash code h,* of fixed length *m*

h = H(M), where *h* is of length *m*

The whole point of the one way hash function is to provide a "fingerprint" of M that is *unique* (Schneier, 1996). The hash code is a function of every bit of the message. This provides an error detection capability since a change in only one bit of the message would result in a change the hash value (Stallings, 1995).

The characteristics of hash functions that make them *one-way* are (Merkle, 1979):

- Given *M*, it is easy to compute *H.*
- Given *H*, it is hard to compute *M* such that *H(M) = h.*
- Given *M*, it is hard to find another message, *M'*, such that *H(M) = H(M').* (collision resistance[1])

A *keyed* variation of one-way hash functions a.k.a. *messages digest* functions can be used for message authentication. The actual ideas for the usage of keyed hash function are old, some of the proposed methods[2] *append*[3] (Tsudik, 1992) (Galvin, 1991) or *prepend*[4] (Galvin, 1991) or even do both (*envelope method*), a secret key (or *two*) to the message before the one-way hash function is applied. The benefit of using message digest functions instead of the previous two methods of authentication, given the hypothesis that confidentiality is not required, is that they are less computationally intensive than encryption, therefore, they are faster. Additionally there are several other

[1] Collision resistance is a very important property of hash-function however it is beyond the scope of this thesis to discuss issues related to that (strong, weak, etc.).

[2] Appearing in SNMPv2.

[3] Secret Suffix Method.

[4] Secret Prefix Method.

reasons indicating that encryption should be avoided – even when it is available in hardware implementation – whenever possible. (Tsudik, 1992) points out them:

- Encryption software is slow, even for a small amount of data.
- Encryption hardware is not extremely expensive but the cost adds up as each node in the network must be equipped with this hardware.
- The overhead to initialize encryption hardware for small blocks of data is very high, mainly because encryption hardware is usually optimized for large data blocks.
- Encryption algorithms are subjected to U.S. export control.
- Some of the algorithms are patented, like the RSA public-key algorithm, and there is an additional cost for licensing them.

Two of the most popular – and fast – one-way hash functions are the MD4 (Rivest, 1990) and its improvement, the MD5 (Rivest, 1992). Both were designed by Ron Rivest, and they produce a 128-bit message digest of the input message. Another famous algorithm is SHA (Secure Hash Function). It was introduced by NIS, along with NSA for use in DSS (Digital Signature Standard)(NIST, 1994). It produces a 160-bits but it is based on principles similar to the MD algorithms. According to (Schneier, 1996) there are no known cryptographic attacks against SHA and because it produces a 160-bit hash value it is more resistant to brute-force attacks than 128-bit hash functions.

An interesting variation of *keyed* hash functions is proposed in (Preneel, 1995). The authors first expose the vulnerabilities of previous proposals of keyed hash functions and then they propose to include a key at the beginning, at the end and in every iteration of the algorithm in order to strengthen it. The new algorithm is called MDx-MAC and the authors claim for the proposed method that:

- It maintains the benefits of keyed one-way hash function usage (minimum implementation effort – maximum confidence).
- The performance would be close to that of a one-way hash function.

- The required resources (memory etc.) for its implementation would be similar to that of a one-way hash function.
- It is generic (can be used with any algorithm with the same principles as MD4).

C. SUMMARY

In this chapter, we discussed computer security issues that are of interest as a background for this thesis. We are interested in particular in two categories of firewalls: *packet filtering firewalls* and *circuit gateways*, and in message authentication using *keyed hash functions*. The next chapter discusses networking issues of interest.

III. HIGH SPEED NETWORKS

A. IP OVER ATM

ATM has been chosen by International Telecommunication Union (ITU-T Recommendation I.121) as the basis for the Broadband Integrated Services Digital Network (B-ISDN). An *Integrated Services* network is one that can be used to carry all types of traffic: data, voice, video etc. "Broadband" means simply *high link speed* (up to Gigabits per second)[1]. ATM networks are being installed in both research[2]and commercial environments[3].

The Internet Protocol (IP) is the foundation protocol for the most widely used packet-switched network, the Internet. The popularity of IP is ever growing with the growth of the Internet.

It is becoming increasingly important to interconnect IP and ATM networks. The easiest way to do so is for IP to treat ATM as a link-level (Layer-2 of the OSI model) technology (like Ethernet or FDDI), ignoring the routing and quality-of-service aspects of ATM. This approach is called *IP over ATM* (Figure 3), and it is still an area of active research. Additionally there is not a universally accepted method for IP over ATM. (Keshav, 1997)

It is beyond the scope of this thesis to examine in depth the different proposals of implementing IP over ATM. However, we think it would be useful, as background, to briefly describe the following:

- The method of IP encapsulation in ATM Adaptation Layer 5(AAL5),
- Cell loss detection Classical IP over ATM (CIP), and

[1] The term comes from the cable television: *baseband* is a single signal used for video, the band of frequencies can be chosen from the base of the spectrum; when many of these signals are modulated and fed into a cable a *broad band* of frequencies are used for all the channels(Kercheval, 1998)

[2] University of Southern California – Advanced Biotechnical Consortium (USC-ABC).

[3] Texaco, McDonalds, Chrysler.

- LAN Emulation (LANE).

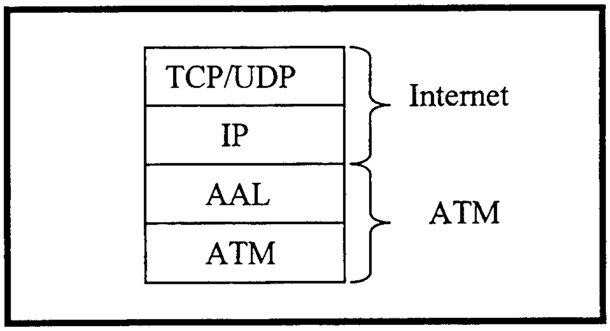


Figure 3. IP over ATM model.

1. IP encapsulation in AAL5

The ATM Adaptation Layer lies on top of the ATM layer Figure 4. AAL adds functionality to the ATM layer, making it more suitable for higher layer protocols or applications. AAL corresponds to the network layer (Layer-3) of the OSI model[1] (Figure 4). Five different AAL versions were developed in order to accommodate the diversity of existing applications. AAL5 is the latest version and possibly, the only one that will be used by future ATM networks. AAL5 provides:

- A mapping of the high-layer data into a cell-stream.
- Cell loss detection.

The higher-layer data are grouped into packets that may have a maximum size of 64 Kbytes. These packets are called *frames.* The AAL5 functionality is achieved in three sublayers:

- The Segmentation and Reassembly (SAR) sublayer distributes the higher-layer data into cells at the sender and reassembles the data at the receiver. It lies directly on top of the ATM layer.

[1] There are arguments that AAL is a datalink layer (Layer-2) however, AAL is a network layer because it provides end-to-end connectivity (even though it does not provide addressing and routing) and it provides a view to the transport layer (Layer-4 of the OSI model) similar to that of a transport layer over IP (Keshav, 1997).

- The Service Specific Convergence Sublayer (SSCS) which lies below the higher-layers. The SSCS is a "shim" layer that fits the upper layer protocols to the AAL. Different applications from higher-layers (e.g., IP, TCP, UDP) send/receive their Protocol Data Units (PDUs) (a.k.a. packets), which "converge" at SSCS.

- The Common Part Convergence Sublayer (CPCS) is the middle layer of the AAL (between SAR and SSCS). The CPCS performs functions common to all protocols.

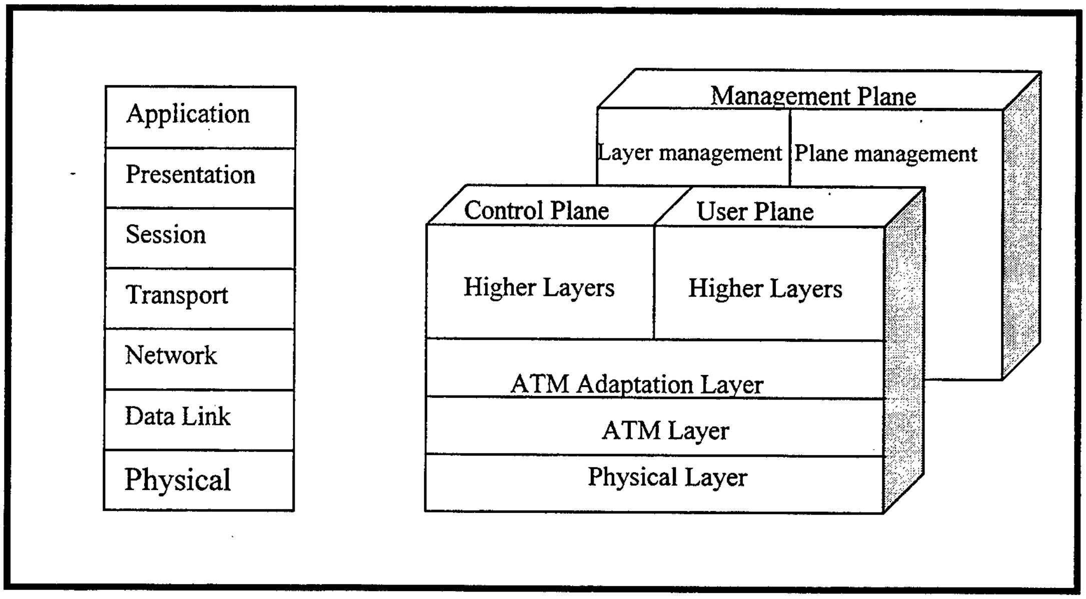


Figure 4. The OSI Model and the ATM Protocol Model. After (Kercheval, 1998)

Below we discuss the two steps required for IP over ATM using the AAL5:

- *Encapsulation of IP datagrams in AAL5 frames*: each IP datagram is put in the data field of an AAL5 frame. The IETF requires an additional *encapsulation layer* to add an 8-byte header to identify which protocol uses the frame. This layer is interposed between the IP layer and the AAL5 layer.(Keshav, 1997)

- *Translation of the IP destination address to an ATM destination address.* There are two major approaches: *classical IP over ATM* and *LAN emulation.* They are described in the next two subsections.

2. Classical IP over ATM

IP (Ipv4) uses 4-byte addresses. ATM uses variable length Network Service Access Point (NSAP) addresses. A protocol similar to Address Resolution Protocol (ARP)[1] could be used for translating an IP address to an ATM one. A problem is that ARP assumes that the LAN supports broadcast, whereas an ATM LAN does not[2]. A solution would be to have an ATM host to play the *ARP server* role. The ARP server translates IP addresses to ATM addresses using a table that contains addresses mappings (from IP to ATM and vice versa) for all the LAN members. When an ATM host wants to send an IP packet to another host in the same LAN, it queries to the ARP server sending its own ATM address and the IP address it wants resolved. The ARP replies with a message containing the address translation. (Keshav, 1997)

The above scheme, when applied to very large ATM networks with thousands of IP/ATM hosts, would create a lot of overhead traffic every time a broadcast (which is common for Internet Protocols) was sent. The classical IP over ATM model partitions the larger ATM network into a set of IP subnets called *logical IP subnets,* or *LISs*, to reduce this traffic. A broadcast from a host can only reach the other hosts on the same LIS. This solution, however, introduces the following problem. A host has no way of knowing if there is a direct ATM path to a host in another LIS. Therefore, in the case where a host in another LIS has to be contacted, the sender must first set up a VC to an IP router. The router accepts the packet and uses IP routing to decide its destination. If the packet is destined to another LIS in the same ATM network, the router opens a VC to the destination (host or another router) and forwards the packet. If the packet destination is not reachable using ATM, the packet is forwarded using normal IP routing. (Keshav, 1997)

The problem with the previous approach is that even if a direct ATM path exists between two hosts, if the hosts belong to different LISs then they are obliged to go

[1] ARP is used to translate IP addresses to datalink Medium Access Control (MAC) addresses.

[2] An ATM LAN is a *nonbroadcast multiple access (NBMA)* LAN.(Keshav, 1997)

through a router every time they wish to establish contact with each other. Katz proposed the *Next Hop Resolution Protocol* in order to address this problem (Katz, 1996). A *Next Hop Server (NHS)* stores IP to ATM address translations for all the hosts (that have registered with it) in the ATM network. The registered hosts are considered part of a logical group called *nonbroadcast multiple access (NBMA) subnetwork* (similar to a LIS). If the IP address can be resolved by the NH server, the destination is contacted over an all ATM path, and IP routing is bypassed (this method of routing is called *cut-through* routing). Otherwise, the packet is forwarded to the local IP router for transmission (as it was done before). (Keshav, 1997)

An additional problem that IP over ATM must address is the translation of multicast IP addresses (a.k.a. Class D addresses). The problem is caused by the fact that ATM is a nonbroadcast technology. A Class D address must be translated to a set of ATM addresses. The process must be dynamic to accommodate changes to the multicast group. The problem can be solved by having an ATM switch create *point-to-multipoint* connections[1]. Two such techniques have been developed:

- The Class D address is associated with a *multicast server* that coordinates the multicast. The server maintains single connections with all the receivers (point-to-multipoint connection). The sender(s) send packets to the server. The server sends the packets to the receivers.

- Each sender maintains a separate point-to-multipoint connection to every receiver in the group. The multicast connection is therefore achieved by a set of point-to-multipoint connections.

In both techniques the translation of a Class D address to a set of ATM addresses is provided by a *multicast address resolution server (MARS)* (Armitage, 1995b). A MARS is an extension of an ARP server with the additional capability to maintain a point-to-multipoint connection – called *cluster control VC* – to every ATM destination in the LIS. The cluster control VC is used to update the mapping from a Class D address to

[1] A virtual circuit that sends cells from a single source to multiple destinations.

a set of ATM addresses as receivers join and leave the multicast group. More details can be found in (Armitage, 1995a).

3. LAN Emulation (LANE)

Continuing with our background information discussion, we will discuss LANE. LAN emulation is the ATM Forum's attempt to make ATM interoperate with the so-called "legacy" networks that include Ethernet and Token Ring (even FDDI). The emulated LAN, however, must be one or the other. A router must be deployed if we want to communication between two emulated LANs. (Kercheval, 1998)

As we have discussed in the previous section, IP over ATM allows IP traffic to be sent to any destination on the ATM LAN or elsewhere simply by translating IP addresses to ATM ones using an ARP server. Classical IP over ATM restricted the destination to be in the same LAN (either a host or a router). It is obvious that if this is the case, the address translation can be done under the data link layer (Layer-2 of the OSI model). The advantage of translating an address under the data link layer is that the address translation is not IP specific. In this case the address translation scheme can be used to carry protocols other than IP over ATM. (Keshav, 1997)

Alternatively, address translation can be achieved by inserting an extra layer, called *LAN emulation client (LEC),* between the data link (MAC) layer and the AAL5 (Truong, 1995). When the LEC layer receives a packet carrying a MAC address, it first checks to see if the MAC address is unicast or multicast. If it is multicast, the packet is sent to a *broadcast server*, which sends a copy of the packet to every host on the LAN. (LANE v2 (AF-LANE, 1997) supports *enhanced multicast,* a form of multicast where not all hosts must receive multicast traffic.) If it is unicast, the ARP cache is checked to see if it already has a translation from the MAC address to an existing VC. If no VC exists, the LEC sends an ARP message to a *LAN emulation server (LES),* requesting resolution of the address. (The LES stores the translation of every MAC address in the LAN.) The ARP responds to the LEC with an ATM address and the LEC uses it to set up a VC. (The LANE endpoint must be capable of ATM signalling (Keshav, 1997).)

Since LEC and LES operate at the data link layer, any network layer protocol can use LANE without modification. ATM, therefore, can replace Ethernet or Token Ring in a straightforward manner (Keshav, 1997). In LANE v2, LECs can request for a particular class of quality of service (QoS)[1] when they set up the VC[2]. That was not possible in v1 where all traffic was treated as belonging in the Unspecified Bit Rate (UBR) class. Detailed information about LANE can be found in (AF-LANE, 1997).

B. TAG SWITCHING

The following is a summary of (Rekhter, 1997), which contains an overview of Cisco Systems' Tag Switching architecture. Tag Switching is a novel approach to Layer-3 (network layer) packet forwarding. The claimed features of the Tag Switching technology are the following:

- The use of the simple *label swapping* forwarding paradigm improves the forwarding performance.
- A tag[3] can be associated with a wide range of forwarding granularities so the same forwarding paradigm can be used to support a variety of routing functions. For example, in one case a tag can be used for destination-based routing (the tag is associated with a point-to-point connection), while in another case a tag can be used for multicast (the tag is associated with a point-to-multipoint connection).
- The simplicity of the forwarding paradigm, the wide range of flow aggregation granularities and the fact that the forwarding paradigm remains the same enables a tag switching system to gracefully evolve to accommodate emerging requirements.

[1] You can have Available Bit Rate (ABR).

[2] A signalling function.

[3] A tag is defining a flow.

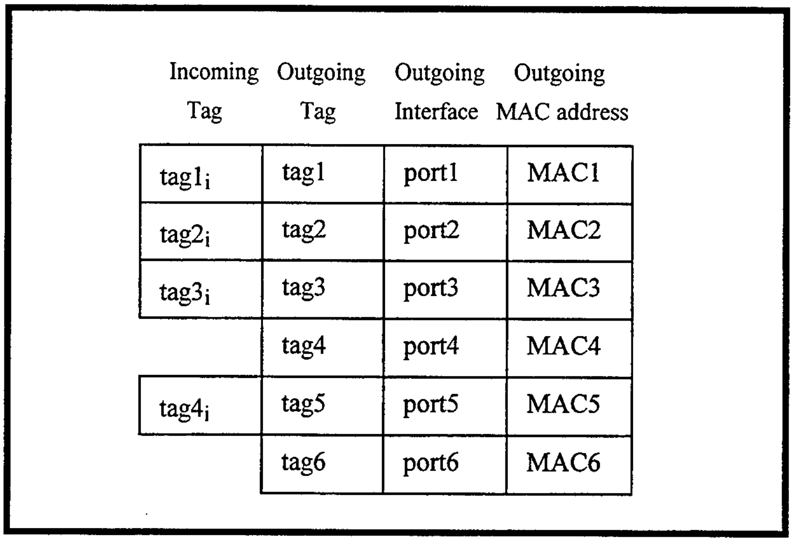
Next, we describe the two main components of Tag Switching architecture: the *forwarding* component and the *control* component.

1. Forwarding Component

When a packet with a tag is received by a tag switch, the switch uses the tag as an index in its *Tag Information Base (TIB)* (Figure 4). The TIB contains entries that consist of an incoming tag, and one or more sub-entries[1] of the form [*outgoing tag, outgoing interface, outgoing link level information*]. The switch looks in the TIB to find an entry that has an incoming tag equal to the tag carried in the packet. If the tag is found, for each sub-entry the switch replaces the tag in the packet with the outgoing tag, the link level information (e.g., MAC address) with the outgoing link level information and forwards the packet to the outgoing interface. The following are the important characteristics of the forwarding component:

- The forwarding decision is based on the *exact match* algorithm using a fixed length, short tag as an index. Therefore the forwarding procedure is simple and allows high forwarding performance (more packets per second). Additionally the simplicity of the forwarding component allows for its implementation in hardware.
- The forwarding decision is independent of the tag's granularity so the same forwarding paradigm can be used to support different routing functions. A good example demonstrating this is the way that the algorithm is used for unicast and multicast. In unicast, an entry in the TIB would only have a single [*outgoing tag, outgoing interface, outgoing link level information*] sub-entry. In multicast, an entry would have one or more sub-entries and the outgoing link information would include a *multicast MAC address.*

[1] In the case of multicast one incoming tag must be associated with many outgoing tags thus we have more than



Incoming Tag	Outgoing Tag	Outgoing Interface	Outgoing MAC address
$tag1_i$	tag1	port1	MAC1
$tag2_i$	tag2	port2	MAC2
$tag3_i$	tag3	port3	MAC3
	tag4	port4	MAC4
$tag4_i$	tag5	port5	MAC5
	tag6	port6	MAC6

Figure 5. Tag Information Base (TIB) format.

- The forwarding paradigm is independent of the routing functionality therefore any new routing functions can be readily.
- The tag-forwarding component is independent from the network layer (Layer-3) and it can be used with various Layer-3 protocols.

A decision has to be made by the designers of a tag-switching network regarding the tag encapsulation in a packet e.g., where to put the tag information. So far, the following possibilities have been proposed:

- A small tag header may be inserted between the Layer-2 and the Layer-3 headers.
- Encapsulation in the Layer-2 header.
- Encapsulation in the Layer-3 (network layer) header (e.g. using the Flow Label field in IPv6).

The flexibility of tag encapsulation allows the implementation of tag switching over a variety of different links, like point-to-point links, multi-access links, and ATM.

one sub-entries.

2. Control Component

In tag switching it is important to be able to identify Layer-3 routes by looking at a tag. This *binding* is essential. In different forwarding granularities a tag may support unicast (the tag identifies one route) or multicast (the tag identifies more than one routes). The control component is responsible for the creation of the tag-route bindings and for the distribution of the tag binding information among tag switches. The control component is modular with different modules supporting different routing functions. The benefit of this modular design is that when new routing functions emerge, tag switching will be able to support them by adding new modules.

Further details of how the control component is supposed to support the routing functions of *destination-based routing, hierarchy of routing knowledge, multicast* and *flexible routing* can be found in (Rekhter, 1997).

3. Tag Switching with ATM

Since both the tag switching forwarding paradigm and ATM cell forwarding is based on label swapping, tag switching technology can readily be applied to ATM switches by implementing the control component of tag switching.

The tag can be carried in the VCI field of the cell. If two levels of tagging are needed, then the VPI field could be used as well[1].

The switch should be able to participate as a peer in network layer routing protocols in order to be able to get the necessary control information for tag bindings. Routing protocols (e.g., OSPF for interior routing, BGP for exterior routing) are used for exchanging information about the network topology – both interior and exterior. This information must be stored in the switch so that it is able to successfully perform the bindings between tags and routes. Additionally, the switch may need to allocate more than one tag for the same route. This is required to avoid packet interleaving when packets arrive from different upstream tag switches that are destined for the same next hop.

[1] One level of tagging (the VCI field) is adequate for most applications

Therefore, the minimum requirements for an ATM switch to support tag switching are: capability to support network layer routing protocols and ability to implement the tag switching control component. An additional requirement may be support for some form of network layer forwarding.

The benefits of implementing tag switching on ATM switches are:

- It would greatly simplify the integration of ATM switches and routers because an ATM switch would appear as another router to an adjacent router. This fact could provide a more scalable alternative to the overlay model.
- It removes the necessity for ATM addressing, routing and signalling schemes since now the forwarding process is topology-driven and not flow based.
- It maintains the ATM switch's ability to support traditional ATM. The tag switching control component and the ATM control plane (the corresponding ATM control component e.g., PNNI[1]) would operate in a non-interactive manner[2].

4. Summary

The following characteristics of tag switching make it an interesting proposal:

- It is not constrained by a particular network layer protocol – it is a multiprotocol solution.
- The simplicity of the forwarding component facilitates high performance forwarding.
- The forwarding component can be implemented in high performance hardware such as ATM switches.

[1] Public/Private Network to Network Interface is the collection of signalling functions for setting up, maintaining, and closing a connection. Another such collection is the User to Network Interface (UNI) that refers to the signalling functions between a single workstation and the network.

[2] To achieve this we need to configure and partition appropriately the switch's resources so that each technology (tag switching and ATM) has enough resources available (like VPI/VCI space etc.).

- Tag switches have no impact on routers since they use the same routing protocols.
- The control component is flexible and may support a variety of forwarding functions (e.g., destination-based routing, multicast etc.).
- A tag can be associated with a wide range of forwarding granularities, which in turn allows for scalability and functionally rich routing.
- The combination of a wide range of forwarding granularities and the ability to evolve the control component independently of the forwarding component make tag switching a technology that enables graceful introduction of new routing functionality. This is very important in a rapidly evolving computer networking environment.

D. SUMMARY

In this chapter, we discussed computer networking issues that are of interest as a background for this thesis. We are interesting in particular in high-speed networks and IP over ATM technologies. We also discussed a novel routing technology, Tag Switching (Rekhter, 1997). In the next chapter, we will examine the problems that this technology introduces and we will present our protocol to address those problems.

IV. THE PROTOCOL

In this chapter, we examine the vulnerabilities introduced by the implementation of the forwarding component of tag switching. In particular, we describe how the security policy of an enclave can be violated by unauthorized access from inside the enclave to the outside world. We further propose a security protocol. The protocol can be applied as a module of the control component in a Network Access Controller (NAC) to minimize these vulnerabilities.

A. INSIDER ATTACKS

Tag switching – and more specifically the *label swapping* forwarding paradigm – introduces security vulnerabilities that have not been fully examined. For example (Rekhter, 1997) does not discuss any security issues.

In this section, we will try to identify a category of vulnerabilities related to the unauthorized use of network resources by an insider. We will examine how an enclave insider (with physical access to systems in the enclave), who has no authorization to send traffic outside the enclave, can exploit the label swapping forwarding paradigm to violate the access restriction imposed upon him/her. This, in turn, will violate the security policy of the enclave.

We focus attention on the forwarding component of tag switching. We do not try to examine the control component because it may differ greatly from implementation to implementation and it may evolve through time. On the other hand, the forwarding component – label swapping – is advertised to be a constant aspect of tag switching. Therefore, we are interested in making the forwarding component more secure.

In this context, we examine which of the attacks described in Chapter II, Section B, can be performed by an insider. We describe the principles on which possible attacks could be based rather than the exact techniques that can be used by an attacker. We believe that several case studies would be required to describe the latter, and they are beyond the scope of this thesis. Additionally, since it is virtually impossible to exhaustively examine all the techniques that a resourceful attacker can come up with, we have decided to take a more generic approach.

Before looking at the possible attacks, we list our assumptions for the enclave:

- There are users in the enclave that do not have access privilege to the outside network and their packets are not allowed to leave the enclave. A situation like this may arise when the enclave enforces an access control mechanism as part of its security policy. The information security officer may decide to grant access privileges to a user and may decide to deny this privilege to another. For example, user Bob has been granted access privilege but user Alice has not. Therefore, Bob is authorized to send traffic out of the enclave but Alice is not authorized to do so and the packets that Alice is sending must be *denied access* to the external network.
- A link encryption scheme is not used (however an end to end scheme may be used). Therefore, the header of a packet goes in the open (but the contents of the packet maybe encrypted).
- The potential internal attacker has spoofing capabilities i.e., (s)he can read the header of any packet.

By examining the traffic in the network, the attacker can see the tags used in the packets sent out of the enclave by authorized users. He or she then can take two possible initial actions to compromise the security policy of the enclave:

- Steal and reuse a valid tag to send unauthorized traffic without notice.
- Alter tags to cause packets to be forwarded to the wrong destination without being detected.

Since the tag is the only routing information used by the forwarding component, a tag switch, upon receiving a packet with a known tag, would forward the packet without further checking[1].

[1] The tag switch looks up the Table Information Base (TIB) to find the incoming tag. Since the tag is valid, an

The remaining difficulty an attacker has to overcome is to ensure that the outgoing packets are going to be received by an intended partner in the outside. This can be achieved (comparatively simply) if the partner uses a sniffer program that picks up packets that leave the enclave. We will not examine how the attackers orchestrate this final stage of the attack. Our focus is how the internal attacker can manage to send packets out of the enclave. By denying this capability, we do not allow the attack to proceed to the final stage.

We consider the following three broad categories of insider attacks. We describe how each of them can be executed by an attacker when the label swapping forwarding paradigm is used in an enclave.

- *Disclosure*: Alice can steal a tag from Bob and start sending packets outside the enclave. This constitutes a direct violation of the DAC policy of the enclave. Additionally, another user authorized to access the network, could copy packets from the traffic sent by Bob, replace the tag with a new valid tag, and then resend them anywhere he or she wishes. Only end-to-end encryption would prevent the actual disclosure of data in the latter case[1].

- *Masquerade*: An unauthorized user can use a stolen tag to insert packets to the network. Additionally, higher layer computer communications protocols would not detect when the attacker steals and reuses the entire IP header. This kind of attack is also known as cut-and-paste attacks.

- *Timing modification*: An attacker can delay or replay packets – belonging to a legitimate user. For a replay of packets whose tags have expired in the switch's TIB, the attacker must steal a new, valid tag and use it for every packet intended to be replayed. The "faked" flow may not have to belong to the original source of the packets. In the case where the attacker needs to replay packets as a flow from a different host, then other parts of the header

entry exists. It simply replaces the incoming tag with an outgoing tag and adds a data link header to the packet. Then the packet is forwarded to the interface(s) designated in the sub-entry(s) of the incoming tag entry in the TIB.

[1] Covert channels are also a problem but they are beyond the context of this thesis.

have to be changed in order to "fool" the higher layers of the communications protocol.

B. THE PROTOCOL

1. Protocol Overview

In order to counter the kind of attacks described in the previous section, we propose the implementation of a protocol that will authenticate every packet that arrives at the tag switch. In this way only authorized packets i.e., packets that come from users with external access privilege, will be forwarded to the outside network. We also want the packet authentication and filtering scheme not to significantly affect the high-speed element of tag switching.

Our protocol requires authorized users to append a fixed-length *trailer* to every packet. This trailer will contain all the necessary information required for the authentication of the packet. The decision to use a trailer was motivated by the following two facts:

- We wanted to avoid parsing the variable-length IP header, which is computationally expensive.
- It is simple to extract a fixed number of bits from the end of a packet.

Our protocol will allow a firewall-like device, called Network Access Controller (NAC), to control the access from an enclave network to the outside. The NAC will play the role of a security "gateway" to the network. The NAC can be implemented at the first tag switch reached by packets on their way out of the enclave or may be a dedicated host at the edge of the enclave. A model for the protocol is shown in Figure 5. The implementation of the NAC will not affect or alter the tag switching technology. It will only add an intermediate authentication stage before the forwarding component of the first tag switch.

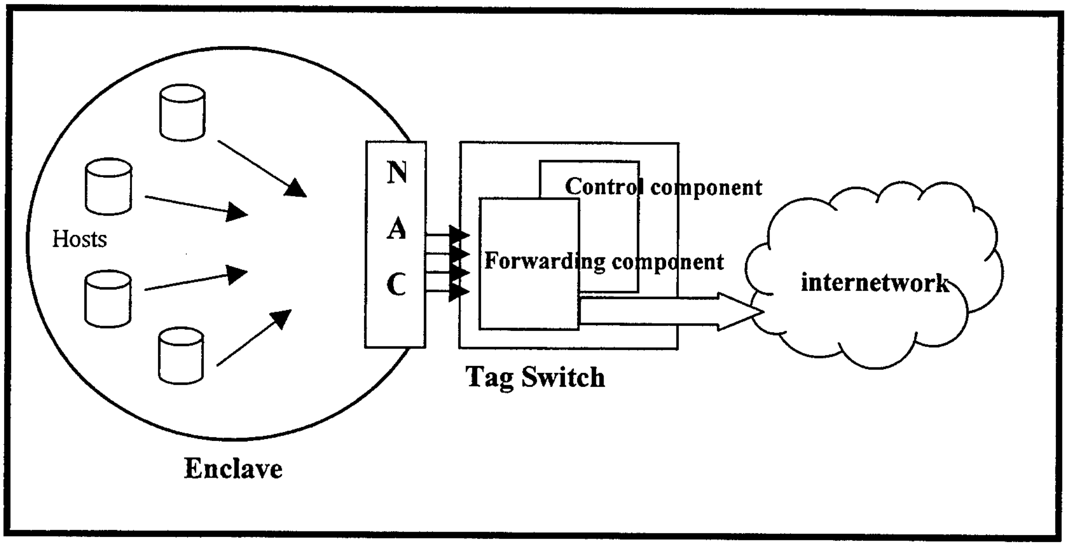


Figure 6. System Model for the Network Access Controller (NAC).

The protocol provides a session-based solution[1]. A session is a period of time that a *user-host* pair has access to the network through the NAC. A session terminates either when the host completes sending its traffic or when the NAC *times-out* the host, requesting reestablishment of the session. The time-out enhances security because it prevents an idle session from being hijacked. The reestablishment of a session does not necessarily mean interruption of the data transmission channel because session reestablishment can happen asynchronously, out-of-band. There will be, in the worst case, only a momentary decrease in throughput. A session may consist of several flows. For example, data flows, voice flows, and video flows may coexist during a single session.

The protocol uses the services of an Authentication Authority to authorize the access to the NAC (and subsequently access to the outside). The protocol also uses the services of a Key Distribution Authority that will provide a *session key* for each user session with the NAC. The Authentication Authority also provides a *ticket* to the user-host initializing this session. The ticket is used by the NAC to verify that the access is

[1] It has to be repeated when a session is established or reestablished. This will become clear in our later discussion of the various phases of the protocol.

legitimate. A detailed description of the verification process is presented in the next section. The NAC creates a *key-table* (an indexed table of keys) for the session. The table contains a sufficient number of keys for the entire duration of the session. A detailed description of the key-table is presented in the next section. A copy of the table is stored in the NAC and another copy is encrypted with the session key and then sent to the user-host. Only the intended user-host has the session key required to decrypt the table.

Before each packet leaves the host a fixed-length MAC for the packet is calculated and appended to the packet. Specifically, when the packet is formatted by the user-host, an appropriate key is chosen from the table (key-table). The key index of the selected key (index of the key in the key-table) denoted by *ki* is inserted into the second[1] word of the trailer (Figure 6). Then MAC is calculated using a one-way hash function on the composite message that consists of the original packet, the first two words of the trailer and the selected 32-bit key. We have made the MAC 128-bits long because we use MD-5 as the default one-way hash function, however, it is easy to change it to match the message digest size of other hash functions. Note that the key is used in the calculation of the MAC; however, it is not included in the packet that is sent. Only the original packet and the MAC trailer will be sent. The original packet payload can be encrypted before MAC computation if high degree of confidentiality is desired. The receiver will decrypt the payload after removing the trailer.

When receiving the packet, the NAC strips the original packet from the trailer and reads the key index (the second word of the trailer) and the MAC (the last 4 words). The next step for the NAC is to check if this is an authorized packet. The NAC looks up the key-table using the key index to find a key[2]. The key should be the same as the key used by the user-host for the MAC calculation. The obtained key is appended to the received packet and the MAC is recalculated at the NAC. If the calculated MAC is the same as the one received in the trailer then NAC knows the packet is authorized and will forward it.

[1] The first word is not used currently by the protocol but it is reserved for future use (version information etc.).

[2] If no key is found the packet will be chopped.

Otherwise the packet is unauthorized and the NAC rejects it. The packet is also rejected if the key is not found in the key-table.

2. List of assumptions

Our design of the NAC protocol is based on the following assumptions:

- The protocol is independent of the network layer technology that is going to be used. However, for simplicity we assume that Ipv6 is used.
- The protocol is independent of the link layer technology that is going to be used. However, we assume that tag switching is going to be implemented over ATM. This will allow us to view the tag as an indicator to the VCI that a flow is using. This notion of a tag is easy to understand and is practical.
- The key indexes in one key-table cannot have duplicates so that each key index corresponds to one and only one key. Therefore, we have a "limitation" of a total of 2^{32} key indexes or a total of 2^{32} keys in one table. While the large size appears to be overkill, the 32 bit-field allows for optimized indexing schemes. Following the VPI/VCI paradigm of ATM, we could use the first 16 bits to identify the flows or hosts (host ids) and the rest of them to identify the keys used for MAC calculation. We will not examine this optimized version, instead we will focus on the basic indexing scheme where all 32 bits are used as key indexes[1].
- An Authentication Server is available in the enclave. Any authentication server can be used for our protocol. We currently assume that the enclave uses the services of Kerberos 5. Kerberos 5 is very popular and already in use in many sites and will be included in Microsoft Windows NTv.5.
- A Key Distribution server is available within the enclave. We choose Kerberos 5 to play this role as well for the same reasons. Additionally,

[1] This scheme is very simple, however, we must keep in mind that it may create a very big aggregate key-table in the NAC (depending on the number of hosts with active flows in the enclave).

Kerberos 5 allows a choice of encryption algorithms (e.g., a client may ask for triple DES, or IDEA and Kerberos 5 will supply the corresponding session key).

- As we mentioned in the previous section the NAC may be the first tag switch or a firewall-like host playing the role of the security gateway[1]. In the latter case the NAC will simply filter and forward packets to the first tag switch. Without loss of generality we assume that the NAC is the first tag switch and that our protocol is going to be implemented on the NAC.
- The security overhead must be kept to a minimum. For this reason, we do not use link encryption for the packets because it is computationally costly. Instead, we have chosen to use a cheap, keyed one-way hash function to calculate the MAC for each packet. (The motivation for using this method of authentication was described in Chapter II.)
- The protocol can use any keyed one-way hash function to compute the MAC. We have selected keyed MD5 in order to demonstrate packet filtering can be done with a minimum security overhead. Our protocol overcomes the weakness of the MD5 algorithm – even the keyed version has been shown to be weak (Preneel, 1995) – with the use of *short lived* keys (e.g., the key has only a limited lifetime, denoted T_0, and it is "never" used again) and *periodic* key-table *refreshing* (we renew the key-table at regular intervals).
- The key-table exchange can be conducted out-of-band and, therefore, we assume that it does not add any significant overhead to the in-band data transmission.
- Periodically the NAC *times-out* a session, requesting a new authentication ticket from the user. This sequence of events can take place out-of-band. In

[1] The NAC does not necessarily introduce a single point of failure. In a distributed environment, we may have more than one NAC to implement our protocol. This will allow redundancy and load balancing in the enclave. The number of NACs an enclave needs is a network design decision.

fact, the user may not even notice that at all, except in the case when he will be prompted to enter a password.

- The user, after logging in at a host inside the enclave, is considered as a single entity, called the *user-host pair*. This is important because we are interested in being able to authenticate both the user and the machine. It is beyond the scope of this thesis to examine how this can be achieved.

3. Protocol Phases

We describe in detail *five* phases of our protocol. The description contains step-by-step actions that take place when the protocol is applied.

a. Initialization phase:

- The user logs in at a host inside the enclave
- The host requests authentication for the *user-host pair* from the Authentication Server (Kerberos 5)
- The user requests a *session key* from the Key Distribution Authority (Kerberos 5) in order to contact the Network Access Controller (NAC). In the request, the host specifies the conventional[1] encryption algorithm to be used (e.g., IDEA[2]).
- The Authentication Server authenticates the user-host and the Key Distribution Authority prepares a session key. A reply is sent back to the user-host pair that includes this *session key* and a *ticket*. The reply is encrypted with the algorithm of choice and with a secret key that is shared by the user and the Authentication Server. The ticket is a Kerberos ticket[3] that contains

[1] A public key algorithm can be used in the case that instead of Kerberos we chose to use X.509-based authentication (ChapterI.A.2).

[2] DES or triple-DES can also be used.

[3] We are interested only for these key components of the key and not the exact form of it. More details for Kerberos can be found in RFC 1510(Kohl, 1993).

the session key, the identity of the client (host), the lifetime of the key and a timestamp (for validity checking). The ticket is encrypted with the NAC's secret key. The host is going to send it to the NAC when it requests to initialize a session. The NAC uses the ticket to verify that the user-host is authorized by the Kerberos to access the outside network.

b. Connecting to the NAC

- The NAC receives the ticket from the host that wants to send traffic (packets).
- The NAC decrypts the ticket using its secret key (the one that is shared only with Kerberos)
- The timestamp and the lifetime of the ticket are checked so that the NAC can decide if the ticket is valid or not.
- If the ticket is not valid the NAC disregards the request (and may audit the attempt for security purposes).
- If the ticket is valid, the session key and the host id are extracted from the ticket by the NAC.

c. Key-table preparation and exchange

- NAC creates a key-table (we describe the format and functionality of the key-table in the next section) that will be shared by the host. In order to simplify the protocol description, we will assume that the host requested activation of a single flow, therefore, the key-table is going to be used for only one flow.
- The new key-table is stored into a database of keys that the NAC maintains for all hosts.
- A copy of the key-table is encrypted with the session key and is sent back to the host.

- The host decrypts the key-table; now both the host and the NAC share the same secret key-table.

d. *Packet formatting at the host*

- Every time a host wants to send a packet to the outside, it chooses a key from the key table. The process of choosing a key and related *rekeying* issues will be examined and described in detail in the next section. The *ki* is stored in the key index field (second word) of the trailer.
- The selected key is temporarily appended to the set (*packet*, *ki*) (first phase of the Secret Suffix Method).
- MD5 is applied to the whole (*packet, ki, key*) and a 128 bit MAC is obtained (second phase of the Secret Suffix Method).
- The created MAC is inserted into the last four words of the trailer (Figure 6).
- The composite message (*packet*, *trailer*) is then sent to the NAC.

e. *Packet verification at the NAC:*

- The message (*packet*, *trailer*) arrives at the NAC.
- The trailer is stripped from the packet, the *ki* is read, and the MAC is buffered.
- NAC looks up the aggregate table in order to retrieve the key that corresponds to the key index read from the trailer. (If the key index is not found the packet is rejected.)
- NAC repeats the Secret Suffix Method using the retrieved key and applying MD5 (repeating the same actions as the host) and creates a MAC.
- NAC compares the created MAC with the received MAC.
- If the MACs are not the same, the packet is rejected, otherwise, the packet is forwarded to the tag switching forwarding component.

It is clear from the above that our protocol is independent from tag switching, and therefore, can be applied to any *label swapping* forwarding architecture. Recall that we do not explore how routes are established in the tag switch because this is part of the control component functionality and beyond the scope of this thesis.

4. Protocol Components

In the following section, we describe the protocol components that were discussed in previous sections and we examine their functionality.

a. Packet Trailer

The packet trailer is a fixed length field appended to the end of each packet. The format of the trailer is shown in Figure 6. The length of the trailer is fixed, 196 bits (24 bytes). The first 32 bits (first word) allow extensibility (e.g., to allow storage of version number or other information) and are not currently used by our protocol. The second word (next 32 bits) is the *ki*. We see two straightforward ways for the use of the *ki*:

- The *ki* is used only for indexing keys in the key-table. This would require the NAC to be able to aggregate all the key-tables that are in use by different hosts in the enclave. Every time a packet arrives at the NAC the whole pool of keys (the aggregate table) must be searched in order to find the corresponding key. It is obvious that in this case the key indexes must be unique. The NAC must keep track of which have been assigned and which have been deallocated (not used any more). The later increases the complexity of the NAC, which is something that we try to avoid. Additionally, if the number of hosts is large, then we can expect the aggregate table to be large. Searching it may incur large delay in the authentication process.
- The *ki* also identifies the host that has created the packet. In this case the first part (e.g., first 16 bits) of the *ki* may play the role of a host identifier and the second part indexes the key in the table for that host. The NAC has to maintain a data structure, which will be a set of entries that will be indexed by

a host identifier. Each entry will be a key-table. In this way, there is no need to aggregate the allocated key-tables.

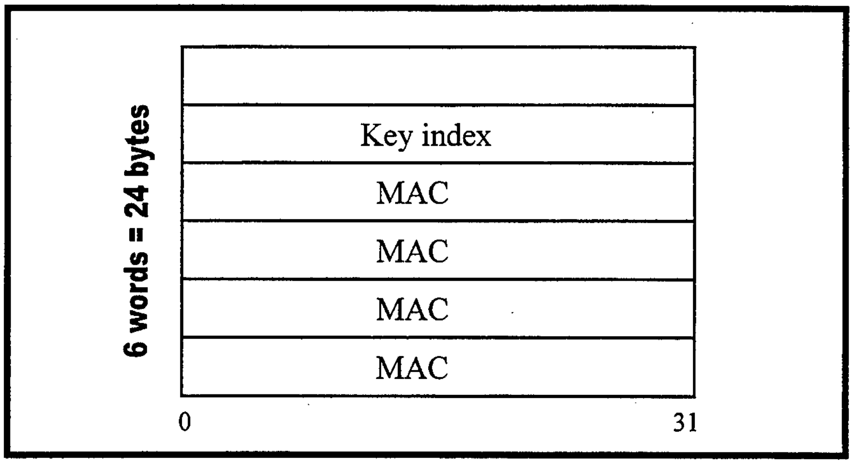


Figure 7. The packet *trailer.*

In this thesis, as we mentioned, we consider the former indexing scheme however, it is obvious that in both indexing schemes the authentication process is totally decoupled with the forwarding process (NAC is independent from tag switching).

b. Key-Table

The key-table is a container data structure that contains (*ki, key*) pairs. The number of entries (pairs) can be decided by the designer. As we mentioned, the table could be an aggregate table for all hosts or it could be an array of tables each of which is specific to a host, indexed by the host_id.

The keys must be randomly selected. The key indexes can be randomly selected too but it is not necessary. The key may be arbitrary long (given that the key-table allows dynamic memory allocation for the key storage). Key generation is not a trivial problem. However, any good random number generator could be used for generating keys.

Another parameter that has to be considered is the *lifetime of the key-table.* This is a design issue. The key-table should be refreshed more frequently if high security is desired. Refreshing the key-table may take place out-of-band, therefore, it does not affect the data throughput for the current session. However there is the synchronization problem for the activation of the new table. The host and the NAC have to work together in order to have a seamless transition from the old key-table to the new. A three-way handshake – similar to the one used for establishing a connection in TCP – between the host and the NAC can be performed for this purpose.

5. Tag-key management

We had to address how the host could select a new key. We refer to it as the *rekeying problem*. Our initial approach was to make the choice of a key random (from the key-table) for each packet. The motivation was that we wanted the lifetime of a key to be the same as the lifetime of a packet and therefore, the lifetime of a packet (comparatively short) would be the time available to an attacker to break the message digest algorithm.

However, this would be true only if our key-table had at least as many entries as the number of packets that a host intends to send. Additionally, a mechanism would be needed to prevent the reuse of a key. In reality, since a key-table has limited length, then it would be impractical not to reuse keys. Key reuse increases the vulnerability to an attack.

Given the above premise (that we have to reuse keys more than once), we examined how long a key is exposed to an attacker with random key selections. In our analysis, we assumed the following:

- The lifetime of the key-table denoted *T*, is the same as the lifetime of the flow (e.g., the table remains the same throughout the transmission).
- The number of entries in the key-table is *n*.
- The flow for the current session consists of *k* packets.

It is obvious that if we use random key selections we allow the whole period *T* to the attacker to break the message digest algorithm. This happens because if a key is selected randomly a new packet may carry any of the key indexes (and subsequently any key) available in the table. Therefore, in this case the *time of vulnerability* is *T*.

Given that a key is selected for each packet, we deduced that *statistically* each *ki* is going to be used *k/n* times. This fact led us to the following observation. If we use the same key, repeatedly, *k/n* times (e.g., put the same *ki* in *k/n* packets) and then discard it we end up using each key index only for *T/n* time instead of T. We decide therefore to use a rekeying timer $T_0 = T/n$. It is obvious that this approach shrinks the time window that the attacker has available to break the message digest algorithm from *T* to *T/ n.* The selection of the next *ki* can be done sequentially[1] from the remaining *unused* key indexes in the table[2]. The only additional action is that every used key index must be marked as *unavailable* in the table or must be totally removed.

Practically, the calculation of *T/n* may not be feasible, so the designer may choose to use a fixed value of T_0 and whenever the key table is exhausted the key-table could be refreshed. The rekeying parameter T_0 is important to our protocol as it widens or narrows the window of vulnerability to the message digest algorithm.

[1] Since the key was produced by a random process, sequential selection of the next entry is acceptable.

[2] We can take advantage of this to optimize the lookup process of the key-table. The NAC could only look up the current and the next entries in the table and not the whole table for matching the key index.

V. PROTOCOL PERFORMANCE EVALUATION

Our protocol introduces delays when it is applied to a network. The major delay is caused by the message digest algorithm (MD5) that we use on every packet. This step is required to create MACs and authenticate packets based on them. Another delay is incurred by the comparison of the recomputed MAC and the received one. Other delays such as the table lookup for key retrieval are very small[1] compared to the MAC calculation and the MAC comparison.

.Any message digest algorithm can be used for MAC generation. We have selected keyed MD5 because MD5 is popular, fast and easy to implement.

A. MD5 PERFORMANCE

Measuring the performance of MD5 is not a new problem. (Bosselaers, 1996) contains some results and a proposal for improving MD5's performance. According to (Bosselaers, 1996) the speed of MD5 (C code for x86 architectures) is 59.7 Mbits/sec on a 90 MHz Pentium. In order to improve the speed the authors rewrote the code in an assembly language and used the following optimizations[2]:

- Keep the code and data in the Pentium's on-chip caches.
- Organize the assembly code to take full advantage of the Pentium's two five-stage pipelines.

The above optimizations increased the throughput of MD5 to 113.7 Mbits/sec. The authors however, mention that "these figures refer to the performance of the hash function's basic building block: the compression function". If one wants to calculate the performance for hashing a message of arbitrary length, he/she has to take into account an additional iteration due to the padding block. (Bosselaers, 1996)

We perform our own measurements for two reasons:

[1] These operations can be optimized to minimize delay. We do not discuss the optimization details because they are beyond the scope of this thesis.

[2] More details on the optimizations can be found in (Bosselaers, 1996).

- To verify their results under a different environment without optimizations.
- To understand how different versions of MD5 are expected to perform.

We measured the throughput of an unoptimized version of MD5 implementation that has used the original MD5 code written in C by Ron Rivest (Rivest, 1992). We downloaded the source code from the Internet (ftp.funet.fi/pub/crypt/hash/mds/md5/). We compiled the code[1] using the Borland C++ 5.02 compiler. The measuring source code is included in Appendix E and the documentation of our[2] code is included in Appendix D. We measured the performance on two machines: a 133 MHz Pentium with 256 Kbytes of Level 1 cache and 48 Mbytes of RAM and a 200 MHz Pentium with 256 Kbytes of Level 1 cache and 48 Mbytes of RAM. During the experiment, we did not take any special measures to optimize the hardware and the software environment of the machines. The OS in the 133 Pentium was Microsoft Windows 95 and in the 200 Pentium was Windows NT 4.0. On both machines, the load was normal.

We applied MD5 to a series of messages of different sizes. The message sizes ranged from 512 bytes to 64 Kbytes. The results for Pentium 133 are shown in Figure 7 and the results for Pentium 200 appear in Figure 8. The measured throughput on the Pentium 133 machine was 35 Mbits/sec, regardless of the message size. On the Pentium 200 machine, the throughput was 60 Mbits/sec. Therefore, our results are in agreement with the performance of MD5 measured in (Bosselaers, 1996).

B. MAC COMPARISON

Another cause of overhead is the comparison of the received MAC taken from the trailer and the MAC that is recalculated by the NAC. Measurements of the comparison process showed that the overhead was negligible when compared to the one incurred by MD5. We tried two different comparison methods. The first method compares the 128-bit

[1] We had to clean the in order to be able to compile it with Borland C++ 5.02.

[2] We had to write a driver and a set of utility functions to interface with the MD5 code.

MAC byte by byte. The second method breaks the 128-bit MAC in four 4-byte words and casts each of them to integers. This results in two groups of four integers. We then subtract each integer from its corresponding peer. If all the (four) subtractions turn out to be *0* then the MACs have to be the same. The results of the two methods are shown in Figure 9 and Figure 10. The documentation and the code are included in Appendixes F and G respectively.

When we compared the two methods for performance, we noticed the following:

- If the MACs are the same, the for-loop that compares the MACs byte-by-byte has to be executed in full. In this case, our measurements show that the byte-by-byte method takes twice the time as the integer method.
- If the MACs are different then the two methods take approximately the same time.

Based on the results, we recommend the use of the integer method for MAC comparison. Note that the delay incurred by the MAC comparison is much smaller than the delay caused by the MD5 calculation; therefore, we can argue that both methods are acceptable.

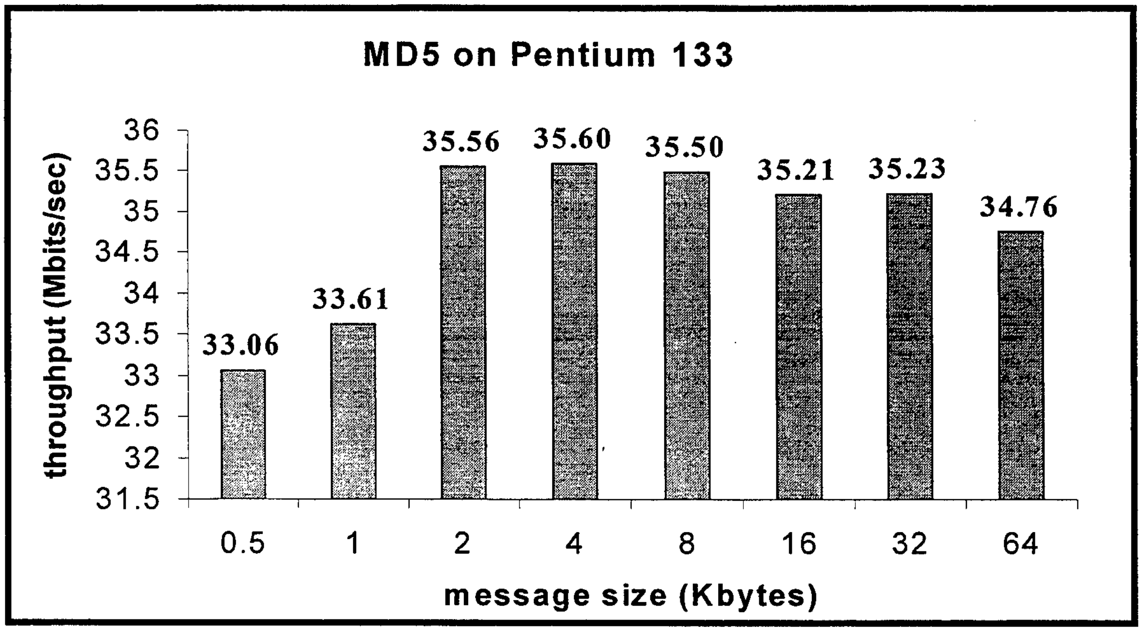


Figure 8. MD5 performance on a Pentium 133 MHz

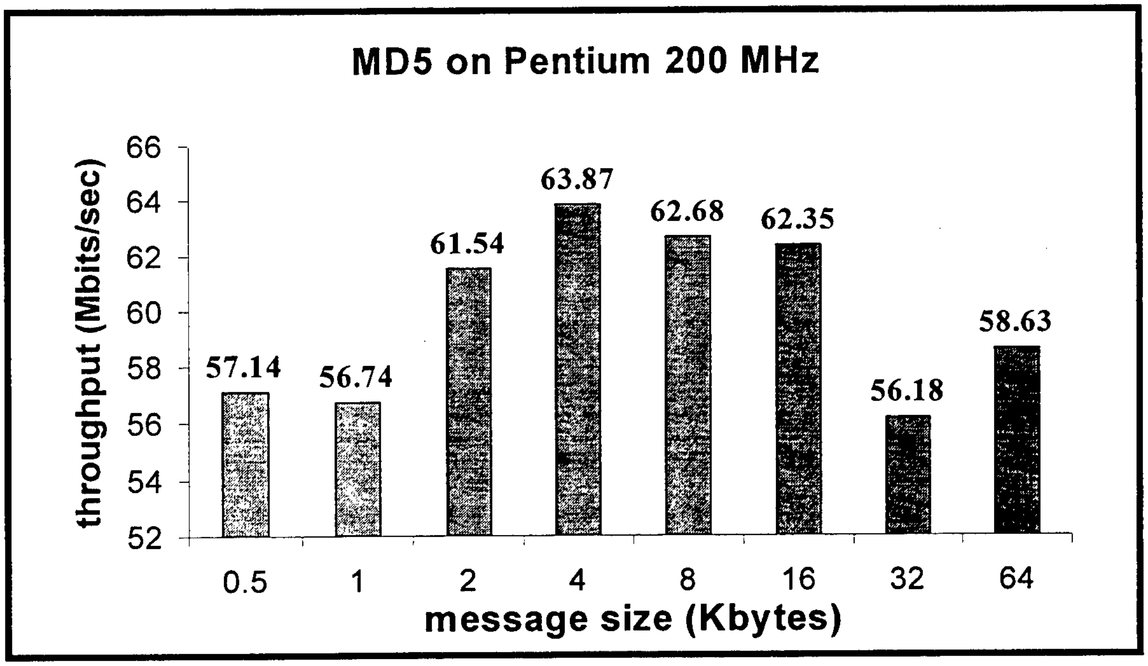


Figure 9. MD5 performance on a Pentium 200 MHz

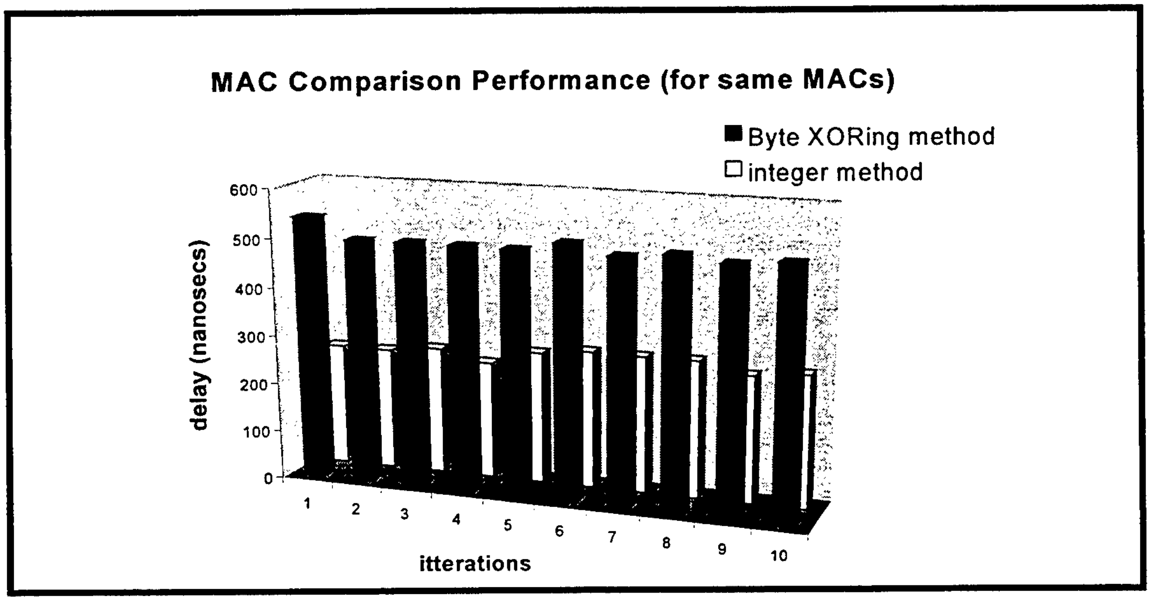


Figure 10. Performance when MACs match.

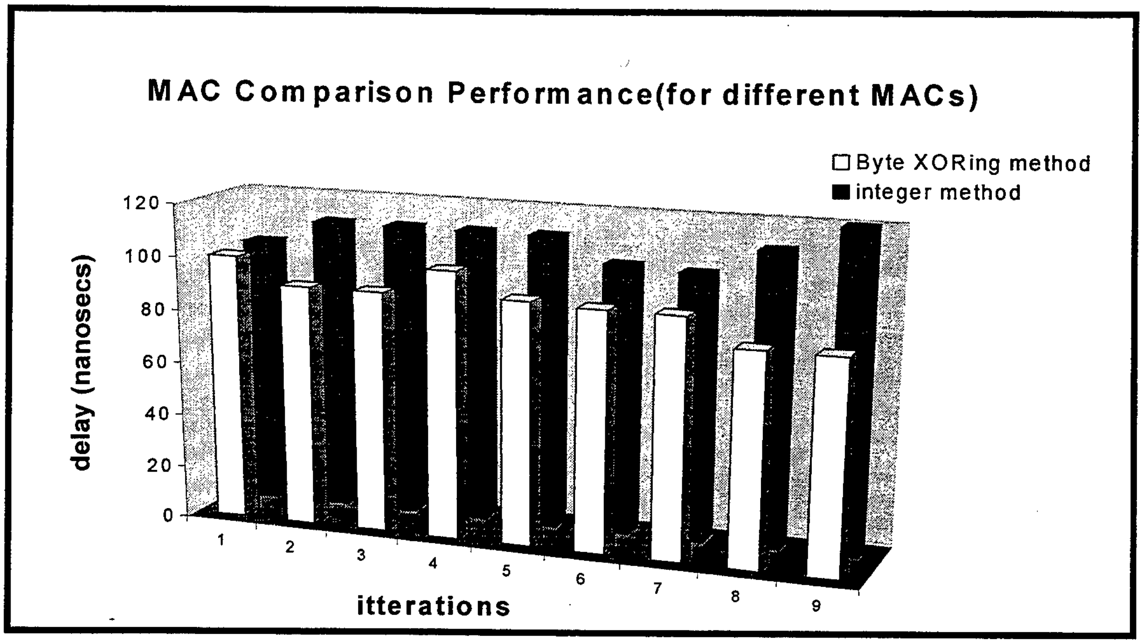


Figure 11. Performance when MACs do not match.

VI. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

In this thesis we examined how the implementation of Tag Switching or other *label swapping,* packet-forwarding, fast-routing technologies increase the vulnerability of an enclave to *insider-attacks* (Chapter IV). This category of attacks is related to unauthorized access from inside an enclave to the outside network. We proposed a protocol to counter this category of attacks. We emphasize that we do not provide protection against *all* kinds of attacks.

The proposed protocol provides the means for fast packet authentication. The high speed is achieved by the following:

- Use of the *trailer*, which allows filtering at Layer 2.
- Use of a cheap and fast message digest algorithm (keyed MD5).
- We use key caching (the host has a table of keys) to allow fast rekeying.

To overcome the weaknesses of the keyed MD5 algorithm we use a limited key lifetime T_0 and periodic *rekeying*. Additionally, we use *key caching*, viz. we provide the host with a key-table, to allow fast rekeying[1]. The performance measurements indicated that it is possible to use our protocol in high-speed networks without unacceptable delays.

The protocol implements a Network Access Controller (NAC). The NAC can be regarded as an authentication module. It can be an intermediate stage between the enclave hosts and the forwarding module of a gateway. If a packet does not make it through the NAC, it will not reach the forwarding module. This can be useful to counter denial of service attacks. The modular nature of the NAC increases its flexibility since it can be easily combined with a variety of routing technologies and other security devices while totally independent of them. The only requirements for the NAC implementation are an

[1] Otherwise rekeying could incur unacceptable delays

Authentication Authority and a Key Distribution Authority, which are usually in place for a concerned network.

B. FUTURE WORK

A *formal verification* of the protocol is required. A prototype implementation in a distributed environment could follow. For the latter, we recommend the following phases:

- Modify the simulation code to use the services of a real Kerberos during the simulation. We recommend this to be done in a Windows NT environment.
- Make the code portable to UNIX machines. In the code, we used C++ Standard Template Library (STL) containers like *maps* and *vectors.* When we were writing the code, Sun's libraries for C++ were not totally compatible with STL. However, this may change in the near future. There are commercially available STLs for Sun machines.
- A Java implementation might be considered. Java has many attractive features that may benefit a prototype implementation of the protocol[1]. Java code is platform-independent, mobile and multi-thread enabled.
- Choose a communication mechanism like *sockets* to pass information back and forth between the different entities in the network, e.g., the host and the NAC. In an initial phase, communication could be simulated over IP. Our code simulates the communication channel using *files* for information exchange. Specifically, each entity (object) that wants to send a key-table creates a file and saves the key-table in the file. The receiving entity opens the file to retrieve the key-table.

[1] Slower performance must be expected with Java.

- Create daemons that will implement the NAC and will provide its services whenever a request arrives. The *nacclass* of our code can be used with little modification for this purpose.
- Simulate the Tag Switching *forwarding component* in software. This can be done if we have an ATM Network Interface Card (NIC) in the machine that implements the NAC. The packet after being authorized by the NAC could be forwarded to a preexisting VC to another machine that could acknowledge its receipt. This will simulate how a tag switch would behave over an ATM link. Knowledge of the ATM Application Programming Interface (API) will be required so that the NIC could be programmed to forward traffic to the preexisting VC.

Another issue that remains to be investigated is the trade-offs between various design parameters. Specifically, experiments can be conducted to evaluate the trade-offs between:

- The lifetime of a key T_0 and security.
- The lifetime of a key T_0 and the number of keys n.
- The key length and the lifetime of a key T_0.

APPENDIX A. DESIGN SPECIFICATION OF SIMULATION PROTOTYPE

1. Description

The simulation program models our protocol. The program will simulate what was described in Chapter IV of this thesis. The platform for the software will be a Pentium-based PC running Microsoft Windows NT 4.0 OS. The language for the simulation is C++ and the Borland C++ 5.02 compiler was used. The object model of the prototype is shown in Appendix B. Future work on this prototype may consist of the phases that were mentioned in Chapter VI.

The program will do the following:

- First it will create a host object, a Kerberos object and a NAC object
- The host object will be registered with Kerberos.
- The host object will request for a session key and a ticket from Kerberos.
- Kerberos will create them (the ticket will be deposited in KEY_TICKET_FILE so the host object will be able to retrieve it).
- The host object requests the start of a session with NAC. It sends the the name of the TICKET_FILE to the NAC. After the NAC verifies it it prepares a tableclass object (the key-table) and deposits it into the TABLE_FILE.
- The host retrieves the table from the TABLE_FILE and starts instantiating Ip_packetclass objects. The packets are formatted in accordance with our protocol using a key from the tablE. The message digest for the set (packet, trailer) is calculated using MD5. The data part of the packet object is retrieved from the contents on the MESSAGE_FILE (this file must pre-exist for the simulation to run; its name must be included in the constants.h file).
- The host requests authentication of each packet by the NAC, following the steps of the protocol. If the NAC authenticates the packet, the program displays that this packet was authenticated.

2. Layers

The file *constants.h* contains most of the system parameters that are used in the simulation. The filenames that are included are required for passing information between objects. The message that is going to be included as the packet content must be in a file, the name of this file must be assigned to MESSAGE_FILE in the constants.h. Without this file the simulation will not run.

The following classes were created (indentation indicates the hierarchy between the modules). We use the expressions host, Kerberos, NAC when we refer to the simulated modules.

- *Hostclass*: models the functionality of a host.
  - *Host_ticket_class*: models the message that a host sends to NAC to establish a session. Prerequisite: the host has been authorized by Kerberos to do so and a valid ticket has been obtained.
- *Nacclass*: models the functionality of a NAC.
  - *Tableclass*: models the functionality of a key-table.
  - *Host_table_vector*: models the database of tables that NAC must maintain.
    - *Host_table_class*: models each entry of the previous database.
- *Kerberosclass*: models the functionality of a Kerberos module.
  - *Host_key_vector*: models the database that Kerberos maintains for hosts and their keys.
    - *Host_key_class*: models the entries of the previous database. Each entry contains the host_id and the secret key that the host shares with Kerberos.
  - *Key_ticket_class*: models the reply of Kerberos to a host that requests to communicate with NAC. It contains a session key and a ticket (intended for NAC).
    - *ticketclass*: models the ticket that Kerberos prepares. The ticket authorizes a host to further establish a session with NAC.
- *Ip_packetclass*: models the functionality of an IP packet.

We also used the following *third party code*:

MD5: RSA Data Security, Inc. MD5 Message-Digest Algorithm (C code). We just cleaned up the code in order to compile and run it with Borland C++ 5.02 compiler.

Finally, we also wrote the following auxiliary code:

Fileutils: provides services, needed by other modules, related to files and buffers.

Hashutils: Services used for interfacing the MD5 package functionality. We modified a demo driver program for MD5. This driver was originally written by Andy Brown (1994, asb@cs.hott.ac.uk). The code was included in the MD5 code that we downloaded from the Internet.

3. Modules

3.1 Hostclass

3.1.1 DESCRIPTION: the class simulates the functionality of a host inside an enclave that wants to get authorization for starting a session with NAC from a Kerberos server. The host needs to obtain a ticket and a session key, establish contact with NAC, obtain a key table and start sending ip_packets. The ip_packet will be formatted in accordance with the protocol we proposed in Chapter IV. We consider the user as part of the hostclass.

3.1.2 DATA:

host_id: an integer, that indicates the identity of the host. A real host id or an IP address could be used in more refined prototypes.

Host_key: a string that is the secret key that a host shares with Kerberos.

User_id: a string that identifies the user to Kerberos. The login name could be used.

ki_key_table: a tableclass object that will be used for getting ki_key pairs for packet formatting. In more refined prototypes a container object (an array or a vector) could be used to contain more than one tableclass objects according to the needs (how many flows need to be active simultaneously) of the host.

Key_ticket_file: the name of the file that contains the session key and the ticket returned from Kerberos. We chose to read them from a file because it is easy to understand and implement. In a more refined implementation we could receive this information from a communication channel established between the host and Kerberos (e.g., a socket buffer or something similar). The simulated ticket is exposed, however this kind of ticket exchange is used only for this simulation.

Ticket_file: the name of the file containing the ticket that the host will send to NAC.

Session_key: this key is going to be used for the ki_key table exchange between the host and the NAC. In a more refined implementation the ki_key table needs to be encrypted with an encryption algorithm using this key.

3.1.3 FUNCTIONS:

Request_authentication

Input: kerberosclass & (a reference to a kerberos module)

Output: the data member key_ticket_file is defined.

Return value: boolean

Description: The host requests from Kerberos authentication by calling the kerberosclass' function that provides this service. Postcondition: the host has the name of the file that contains a session key and a ticket. If the authentication process fails then the return value is False.

Send_ticket

Input: none

Output: HOST_TICKET_FILE contains a host_ticket_class object.

Return value: char*

Description: The host creates a file containing the information that is necessary to NAC for host authentication. Postcondition: the

HOST_TICKET_FILE contains the host_id and a ticket (a host_ticket_class object). The name of this file is returned.

Get_table

Input: const char* TABLE_FILE_NAME

Output: the ki_key_table data member of the host is defined.

Return value: none

Description: The host opens the TABLE_FILE_NAME file and gets the ki_key_table that was prepared by NAC. The host now has a copy of the key table available.

Create_message

Input: char* message_file

Output: an ip_packetclass object is instantiated.

Return value: ip_packetclass

Description: creates and returns an ip_packetclass object. The ip_packetclass object is formatted by applying our protocol and using a ki_key pair chosen from the ki_key table. The *MESSAGE_FILE* name is included in the ip_packetclass object to simplify the process of reading the packet data. BE CAREFUL not to forget to create a message file and put its name in the constants.h file before starting the simulation. A new key is chosen every T_ZERO time.

Get_session_key

Input: none

Output: the date members session_key and ticket_file are defined.

Return value: none

Description: extracts the session key and the name of TICKET_FILE from key_ticket_class object that Kerberos created.

Postcondition: the data members session_key and ticket_file get values.

3.2 **Host_ticket_class**

3.2.1 DESCRIPTION: the class creates objects that contain the host_id and the name of the TICKET_FILE that contains the host's ticket. The objects are created by hosts. NAC uses them for host authorization.

3.2.2 DATA:

host_id: an integer, that identifies the host.

Ticket_file: a string, that is the name of the TICKET_FILE.

3.2.3 FUNCTIONS:

Host_id_is

Input: none

Output: none

Return value: int

Description: The host_id data member is returned.

Ticket_file_is

Input: none

Output: none

Return value: char*

Description: The ticket_file data member is returned.

3.3 **Nacclass**

3.3.1 DESCRIPTION: the class simulates the functionality of a Network Access Controller (NAC). NAC checks that a host (a user-host pair) requesting a session is authorized by Kerberos to do so. The NAC gets the ticket that Kerberos created when the host asked for authorization and extracts the ticket and the session key. NAC then creates a tableclass object (the ki_key table) for the session with the host. Stores the ki_key table in a database and sends a copy back to the host. In a more elaborate simulation, this copy must be encrypted with the session key. When the host starts sending ip_packets, the NAC must play its role in accordance with the protocol described in Chapter IV. Therefore, NAC "strips" the trailer from the ip_packetclass object, reads the ki and calculates the MAC for that ip_packet. If the calculated MAC is the same with the one that was received in the packet trailer, then the packet is authenticated and can be forwarded.

3.3.2 DATA:

hosts_tables_db: the database of hosts and their tables. It is a host_table_vector object (a detailed description follows).

3.3.3 FUNCTIONS:

Prepare_table

Input: char*

Output: a tableclass object is instantiated and deposited in TABLE_FILE.

Return value: char*

Description: The name of the TICKET_FILE that contains the ticket of a host must be passed in as input. If the ticket is valid, NAC instantiates a tableclass ofbject and deposits it in the TABLE_FILE. The function returns the name of the TABLE_FILE. In the opposite case, the string "invalid_ticket" is returned. (Note: this can crash the program, if a function tries to open a file with name "invalid_ticket".)

Authenticate_packet

Input: ip_packetclass

Output: none

Return value: boolean

Description: an ip_packetclass object is passed in by value (a local copy of the object is created). The ki is read from the trailer and the MAC is calculated. If the calculated MAC is the same with the MAC that is stored in the ip_packetclass object trailer the function returns True, otherwise False.

3.4 Tableclass

3.4.1 DESCRIPTION: the class simulates the functionality of key table.The table is created by the Network Access Contoller (NAC) and is sent to a host. We chose to use the Standard Template Library *map* container for the key-table data structure. The map contains (ki, key) pairs. The "first" (the index) of each pair is the ki and the "second" (the value) is the key. We chose the key to be an integer value created by the "srand ()" function of C++, as *seed* we use the host_id. This approach is suitable for demo only. In a more refined prototype, a more robust random number generator should be used. A key_generator could also be used. Additionally, keys could be chosen to be other than integers. The length of the class is defined by the constant NUM_PAIRS. The constant is defined in the constants.h file and we defined it to be 10. The kis are integers as well. We use integers in the range [1- NUM_PAIRS]. The kis could be chosen randomly without affecting the map implementation, but this is not necessary.

3.4.2 DATA:

ki_key_table: a map of NUM_PAIRS ki_key pairs (pairs of integers).

3.4.3 FUNCTIONS:

Get_next_ki

Input: none

Output: none

Return value: int

Description: the next ki in sequence in the key-table is chosen and returned to the caller.

Get_next_key

Input: int

Output: none

Return value: int

Description: a ki is passed in and the corresponding *key* is returned.

Initialize_the_key_pool

Input: int

Output: the tableclass is populated with NUM_PAIRS (ki, key) pairs.

Return value: none

Description: a host_id is passed in. This is used as seed and the values of the keys are randomly generated by the srand function. Postcondition: the ki_key table for that host is initialized.

Look_up_table_for_key

Input: int

Output: none

Return value: int

Description: a ki is passed in and the fuction looks up the map for the corresponding *key* that is returned.(Note: the program will crash if a non-existing tag is passed as an argument.)

3.5 **Host_table_vector**

3.5.1 DESCRIPTION: the class simulates the database of tables that is maintained by NAC. NAC uses this object to keep track of hosts and tables. We chose the Standard Template Library *vector* container to implement the database. The main reason was the flexibility of vectors (a detailed description follows). We can dynamically add more entries and remove old ones without caring for memory management. The vector elements are host_table_class objects (we explain their functionality in detail later). In this simulation, a host uses only one table. This can change easily in prototypes that are more refined, where a host can be associated with more than one table.

3.5.2 DATA:

Hosts_tables: the database, a vector of host_table_class objects.

3.5.3 FUNCTIONS:

Add_host_table

Input: a host_table_class object

Output: the database gets a new tableclass object.

Return value: none

Description: a copy or the host_table_class object is inserted in the vector.

Remove_host_table

Input: int

Output: a tableclass object is removed from the database.

Return value: none

Description: a host_id is passed in and the corresponding entry in the vector (the host_table_class object) is removed.

Is_host_authorized

Input: int

Output: none

Return value: boolean

Description: a host_id is passed in and the (vector) database is searched to find if an entry exists for this host. If a host_table_class object is found with the same host_id the function returns True, otherwise False.

Key_for_host_ki

Input: int, int

Output: none

Return value: int

Description: a host_id and a ki are passed in. The function first finds in the database the corresponding to the host table , and then fetches the key indexed by ki. This key is returned.(Note: the program will crash if no entry exists in the database for the host_id or if a non-existing ki is passed as a second argument.)

3.6 Host_table_class

3.6.1 DESCRIPTION: the class objects contain a host_id and a pointer to a tableclass object. Objects of this class are used as entries in the host_class_vector database maintained by NAC.

3.6.2 DATA:

Host_id: an integer that identifies the host.

Tableptr: a tableclass * pointing to the ki_key table that NAC created for the host.

3.6.3 FUNCTIONS:

Host_id_is

Input: non

Output: none

Return value: int

Description: returns the data member host_id, an int.

Table_is

Input: none

Output: none

Return value: tableclass *

Description: returns a *handle* (pointer) to the tableclass object pointed by the tableptr data member.

3.7 **Kerberosclass**

3.7.1 DESCRIPTION: the class simulates the Kerberos server. It has two databases, one for users (a user vector) and one for hosts and keys (host_key vector). A user or a host can register with Kerberos (enter the database) or can request authentication to begin a session with NAC. In the latter case, the Kerberos will look if the host or the user exist in its databases. If their user_id and the host_key entries are found, then a session key and a ticket are prepared. The session key in our case is stubbed to "sesskey". The ticket is a ticketclass object. The functionality of tickets has been simplified and we don't include timestamps (that real tickets have).

3.7.2 DATA:

Host_key_database: host_key_vector object, a database.

User_database: a vector of char* (user ids).

3.7.3 FUNCTIONS:

Register_host

Input: int, char*

Output: the host_key_database gets a new entry.

Return value: none

Description: an int (host_id) and a char* (key) is passed in and an entry is created in the Kerberos database. Postcondition: the host is now registered in Kerberos. Additionally,Kerberos has a commonly known key with the host.

Remove_host

Input: int

Output: an entry from the host_key_vector is removed.

Return value: none

Description: a host_id is passed in and the corresponding entry in the database is removed.

Register_user

Input: char*

Output: the user database gets a new entry.

Return value: none

Description: a char* user_id is passed in and an entry is created in the Kerberos user database. Postcondition: the user is now registered in Kerberos.

Remove_user

Input: char*

Output: an entry is removed from the user database.

Return value: none

Description: a user_id is passed in and the corresponding entry in the database is removed.

Request permission

Input: char*, int

Output: none

Return value: boolean

Description: a user (char*) at a host (int) are passed in and the vector database is searched to find them. If entries for both exist, then return True, otherwise False.

Send_key_ticket

Input: int

Output: the KEY_TICKET_FILE contains the session_key and the TICKET_FILE name.

Return value: const char*

Description: the KEY_TICKET_FILE is created. The file contains the session key and the name of the TICKET_FILE. The function returns the name of the file.

Create_session_key

Input: none

Output: none

Return value: char*

Description: returns the string "sesskey".

Create_ticket

Input: int, char*

Output: a ticketclass object is instantiated.

Return value: ticketclass

Description: a ticketclass object is instantiated and returned

3.8 **Host_key_vector**

3.8.1 DESCRIPTION: the class simulates the database of hosts and their corresponding keys that Kerberos maintains. Kerberos uses this object to keep track of the hosts and their keys. We chose the Standard Template Library *vector* container to implement the database. The vector elements are host_key_class objects (a detailed description follows). This simulates the database of secret keys that a real Kerberos server maintains.

3.8.2 DATA:

Hosts_keyes: the database, a vector of host_key_class objects.

3.8.3 FUNCTIONS:

Add_host

Input: int, char*

Output: a host_key_object is instantiated. The database host_key_vector gets a new entry.

Return value: none

Description: an int host_id and a char* (secret) key is passed in and the function creates a host_key_class object and adds it as an entry in the database. Postcondition: Kerberos has a secret key for this host.

Remove_host_key

Input: int

Output: an entry is removed from the host_key_vector (the database)

Return value: none

Description: a host_id is passed in and the corresponding entry in the vector (host_key_class object) is removed.

Is_host_registered

Input: int

Output: none

Return value: boolean

Description: a host_id is passed in and the database is searched to find if an entry exists for this host (subsequently, if a secret key is shared by Kerberos and the host). If a host_key_class object is found the function returns True, otherwise False.

3.9 Host_key_class

3.9.1 DESCRIPTION: the class objects contain an int host_id and a char* host_key. The host_key is supposed to be a secret key known only to the host and the Kerberos. Objects of this class are used as entries in the host_key_vector database maintained by Kerberos.

3.9.2 DATA:

Host_id: an integer that identifies the host.

Host_key: a char* that is the secret key known to Kerberos and the host.

3.9.3 FUNCTIONS:

Host_id_is

Input: none

Output: none

Return value: int

Description: returns the data member host_id, an int.

key_is

Input: none

Output: none

Return value: char *

Description: returns the data member host_key, a string

3.10 **Key_ticket_class**

3.10.1 DESCRIPTION: the class objects contain a session key and the name of the file that contains the ticket. Objects of this class are created by Kerberos and are sent to the host.

3.10.2 DATA:

Session_key: a char* (the string "sesskey")

Ticket_file: a char *(the filename).

3.10.3 FUNCTIONS:

Session_key_is

Input: none

Output: none

Return value: int

Description: returns the data member session_key, a string.

ticket_file_is

Input: none

Output: none

Return value: char *

Description: returns the data member ticket_file, a string.

3.11 **Ticketclass**

3.11.1 DESCRIPTION: the class objects contain a session key and a host_id (of the host that requested a session). Objects of this class are created by Kerberos and are sent to the host. The host in turn is going to send them to the NAC. (In a more refined prototype, real life Kerberos tickets can be used).

3.11.2 DATA:

Host_id : an int

Session_key: a char *(the string "sesskey").

3.11.3 FUNCTIONS:

Session_key_is

Input: none

Output: none

Return value: int

Description: returns the data member session_key, a string.

Host_id_is

Input: none

Output: none

Return value: int

Description: returns the data member host_id, an int.

3.12 **Ip_packetclass**

3.12.1 DESCRIPTION: the class objects simulate ip_packets. Objects of this class are created by hostclass objects and are sent to the NAC for authentication.

3.12.2 DATA:

Ki : an int, the key_index of the key that was used for MAC calculation for this packet.

Host_id: an int that identifies the host.

filename: a char*, the name of the MESSAGE_FILE that contains the data that we want to send with this packet.

Mac: a unsigned char*, the message digest.

Note that the ki and the MAC consist the trailer of the packet.

3.12.3 FUNCTIONS:

mac_is

Input: none

Output: none

Return value: unsigned char*

Description: returns the data member mac, an unsighned char*.

ki_is

Input: none

Output: none

Return value: int

Description: returns the data member key_index, an int.

filename_is

Input: none

Output: none

Return value: char*

Description: returns the data member filename, a string. This file contains the data that we want to send with this packet.

host_is

Input: none

Output: none

Return value: int

Description: returns the data member host_id, an int. It is the id of the host that created this packet.

3.13 **Fileutils**

3.13.1 DESCRIPTION: the file contains auxilliary functions for copying files to files and copying files to string streams buffers.

3.13.2 DATA:none

3.13.3 FUNCTIONS:

filecopy

Input: char* fromfile

Output: char* tofile

Return value: none

Description: two filenames are passed in and the first file contents are copied and ovewrite the contents of the second file.

Copytobuffer

Input: char* fromfile, char* tobuffer, int key_index, int key

Output: none

Return value: char*

Description: a file name is passed in and its contents are dumped in the tobuffer string. Additionally the key_index and the key integers are appended to the string. This is protocol specific and a simulates the appending of a packet trailer.

3.14 **Hashutils**

3.14.1 DESCRIPTION: function that allow usage of the MD5 package are written here. The original code as it was mentioned before freely available in the Internet. The code is written in C and it was modified accordingly to accommodate the needs of our simulation.

3.14.2 DATA: n/a

3.14.3 FUNCTIONS:

MD5File

Input: char*

Output: none

Return value: unsigned char* (the 16 byte digest)

Description: a string, the name of a file that we want to calculate its contents MD5 is passed in. The message digest is calculated and returned

MD5String

Input: char*

Output: none

Return value: unsigned char*

Description: a string that we want to calculate its MD5 is passed in. The message digest is calculated and returned

MD5Print

Input: unsigned char*

Output: none

Return value: none

Description: the message digest is passed in, the function prints it as series of hex numbers.

MD5TestString

Input: char*

Output: none

Return value: none

Description: a string, that we want to calculate its MD5 is passed in. The message digest is calculated but it is not returned. This function is used only in the performance measurements of MD5.

compareDigest

Input: unsigned char*, unsigned char*

Output: none

Return value: boolean

Description: two message digests are passed in and are compared. The method of comparison here is byte by byte XORing. The function returns "true" if the message digests are the same. "False" otherwise.

Compare2Digest

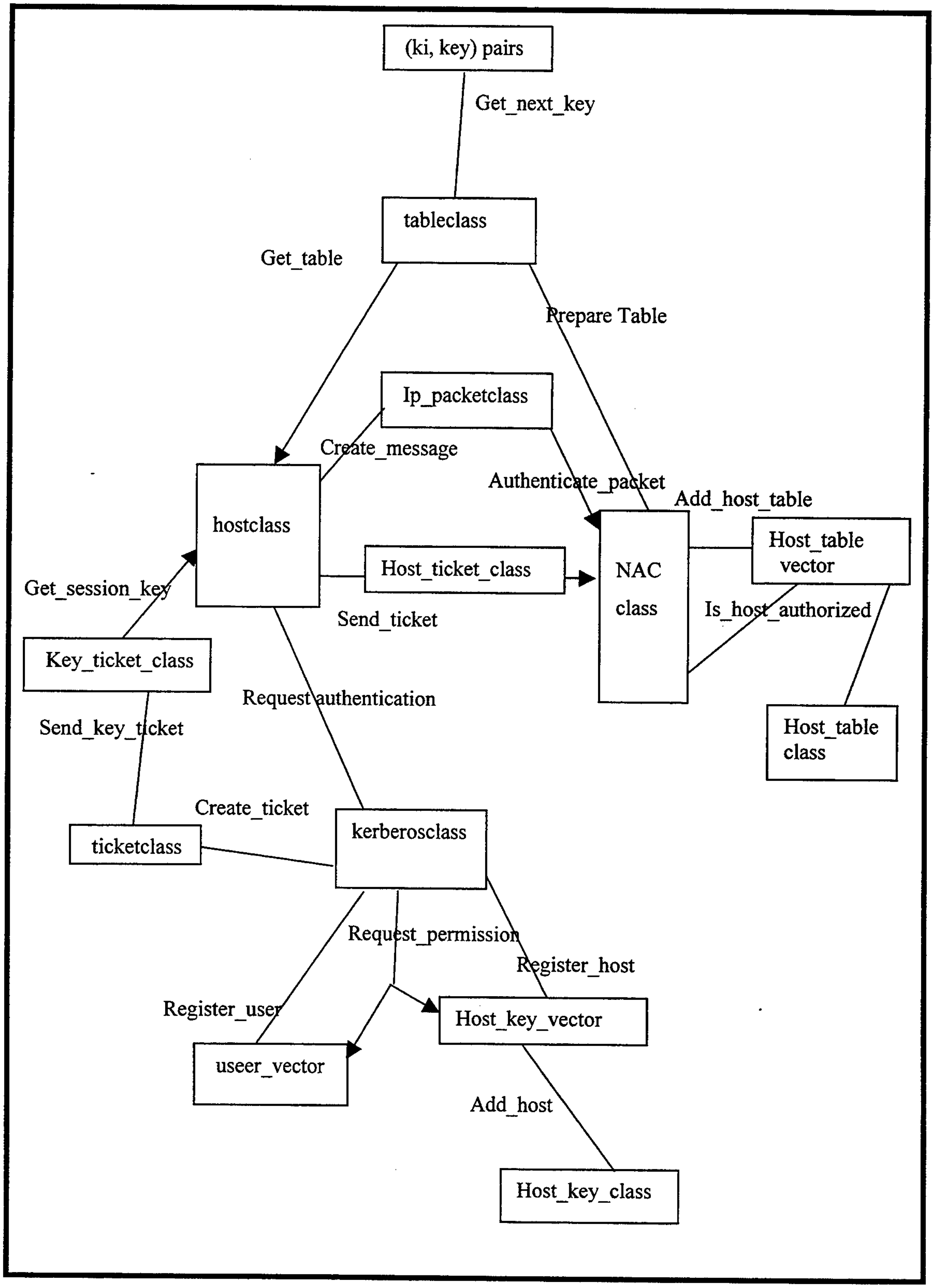
Input: unsigned char*, unsigned char*

Output: none

Return value: boolean

Description: two message digests are passed in and are compared. The method of comparison here is the so called integer methode. We cast every four bytes of the 16 byte digest to integers and we subtract the corresponding ones. If all the results are 0 then the digests are the same. The function returns "true" if the message digests are the same. "False" otherwise.

APPENDIX B. THE OBJECT MODEL



APPENDIX C. SIMULATION SOURCE CODE

```
//*****************************************************************************
//*****************************************************************************
// File : constants.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: Windows NT 4.0
// compiler: Borland C++ for Windows, ver. 5.02
// Date: 17 Sep 1998
// Description: Definitions and consts used throughout the project
//          This file contains the necessary file names that the simulation uses as “communication
//          means. They have to exist for the simulation program to run.
// Assumptions:none
//
// Warnings: none
//
//*****************************************************************************
#ifndef __constants_H__
#define __constants_H__

const char* TABLE_FILE = "keytable.dat";
const char* TICKET_FILE = "ticket.dat";
const char* KEY_TICKET_FILE = "keytick.dat";
const char* MESSAGE_FILE = "message.dat";
const char* FILE_TO_HASH = "hmessage.dat";
const char* HOST_TICKET_FILE = "hosttick.dat";

//for this implementation we choose a table of 10 pairs
const int NUM_PAIRS = 10;
// T_ZERO is the cryptoperiod for the key here we choose 5 msecs
const int T_ZERO = 5;

const int MAX_MESSAGE_SIZE = 1024;

#endif
```

```
//*******************************************************************************
//*******************************************************************************
// File : driver.cpp
// Name : Ioannis Kondoulis
//
// Operating Enviroment: Windows NT 4.0
// compiler: Borland C++ for Windows, ver. 5.02
// Date: 17 Sep 1998
// Description: This is the driver program that executes the simulation.
// Assumptions:none
//
// Warnings: none
//
//*******************************************************************************
//
#include "hostclass.h"
#include "kerberosclass.h"
#include "nacclass.h"
#include <iostream>
#include "constants.h"

void banner();

int main()
{
  bool host_authenticated;

  banner();

  //the players
  kerberosclass kerberos;
  hostclass host(1,"hostkey","ioannis");
  nacclass nac;

  //programmer registers the host with kerberos(hardcoding)
```

```
kerberos.register_host(1,"hostkey");
kerberos.register_user("ioannis");

//the host requests authentication from kerberos
host_authenticated = host.request_authentication(kerberos);

//host "sends" ticket to nac
char* ticketfile = host.send_ticket();

//nac prepares the key table
char* tablefile = nac.prepare_table(ticketfile);

//the host gets the table
host.get_table(tablefile);

//the host sends the ip_packets to nac
//the nac is called to authenticate each packet
cout<< "Press any key to start sending packets."<<endl;
for (int i=0 ; i<10; i++){
  cin.get();
  ip_packetclass packet= host.create_message("message.dat");
  cout<< "Press any key to send the next packet.\n"<<endl;
  if (nac.authenticate_packet(packet)){
    cout<<"Packet " << (i+1) <<" has been authenticated by NAC."<<endl;
    //delete for memory economy
    packet.~ip_packetclass();
  }
  else{
    cout<<"NAC could not authenticate the packet."<<endl;
    //delet for memory economy
    packet.~ip_packetclass();
  }

}
```

```
  cout<<"***************************************\n"
      <<"**            Press any key to <exit>             **\n"
      <<"***************************************\n"<<endl;
  cin.get();

  return(0);
}//end main


void banner()
{
 cout<<"****************************************************************"<<endl
   <<"*                                                              *"<<endl
   <<"*      SIMULATION PROGRAM FOR THE NAC PROTOCOL                 *"<<endl
   <<"*             Author: Ioannis Kondoulis                        *"<<endl
   <<"*                                                              *"<<endl
   <<"*      DISCLAIMER: this program is for demostration ONLY!      *"<<endl
   <<"*        for any other use the author does not have any        *"<<endl
   <<"*        responsibility.                                       *"<<endl
   <<"*                                                              *"<<endl
   <<"****************************************************************"<<endl;

 cout<<"PROGRAM LIMITATIONS: The program will create and register only one \n"
   <<"      simulated user. The simulated user will create and send 10\n"
   <<"      packets to the NAC. After its packet is authenticated it will \n"
   <<"      be displayed on the screen with a prompt to hit a key to  \n"
   <<"      send the next packet. The program is not robust and its behaviour \n"
   <<"      is not guaranteed for sending more than these packets, to do so further  \n"
   <<"      work and testing is needed.                          \n"
   <<endl;

  return;
}
//end driver.cpp
```

```
#ifndef __fileutils_H__
#define __fileutils_H__

//************************************************************************************
//************************************************************************************
// File : fileutils.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:the file contains a file copy function
//
// Assumptions: the max values do not exceed the consts declared here
//
// Warnings:be careful when you pass paths '/' means ignore the next need to
//        use two "//" for the path to be valid
//        TO DO doublecheck the above!!
//
//************************************************************************************
//
#include <iostream>
//#include <string>
//#include <stdlib>
#include <strstream>
#include <fstream>

//used for file names
const short MAX_STRING_SIZE = 40;

//for each line in the file
const short MAX_LINE_SIZE=256;

//if you want to restrict the lines of the file ADD
```

```
//no more of so many lines in a file
//const short MAX_NUM_LINES=512;

//the copy function expects two filenames
void filecopy(char* fromfile, char* tofile);

//the copy to a buffer function expects a filename, a pointer to a string
//and the key that will be appended to the buffer contents
char* copytobuffer(char* fromfile, char* tobuffer,int key_index, int key);

#endif

//end fileutils.h file
```

```
//****************************************************************************
//****************************************************************************
// File : fileutils.cpp
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:the file copy function
//
// Assumptions: the same as in fileutils.h file
//
// Warnings: When the driver is modified to create more than 10 packets the program breaks in the
//          copytobuffer function in the marked BREAKPOINT. The driver works fine as it is now,
//          if you want to modify it you must increase the robustness of the code, especially memory
//          management issues and include exception handling mechanisms.
//****************************************************************************
//
#include "fileutils.h"

//------------------------------------------------------------------------
//function: filecopy(char* fromfile, char* tofile)
//return value: none
//parameters: char* fromfile, char* tofile
//purpose: copies line by line the contents of fromfile to tofile
//       The do while executed at least once so it can read the eof of an
//       empty file
//------------------------------------------------------------------------
//
void filecopy(char* fromfile, char* tofile)
{
  char* strLinePtr;
  char* tempLinePtr;

  //CASE IF YOU READ THE fromfile FILENAME FROM STD INPUT
```

```
//no need for now
//char* tempname;
//tempname = new char[MAX_STRING_SIZE];
//cin>>tempname;
//fromfile = new char[strlen(tempname)+1];
//strcpy(fromfile,tempname);

//CASE IF YOU READ THE tofile FILENAME FROM STD INPUT
//no need for now
//cin>>tempname;
//tofile = new char[strlen(tempname)+1];
//strcpy(tofile,tempname);
//free the tempname memory
//delete[]tempname;

//copy the contents of fromfile to tofile
//open the two files appropriately
fstream in(fromfile, ios::in);
fstream out(tofile, ios::out);
//allocate temporary memory for contents of each line (buffer)
tempLinePtr = new char[MAX_LINE_SIZE];

do{//read each line
  in.getline(tempLinePtr,MAX_LINE_SIZE,'\n');
  //allocate exactly so many memory as needed tor this line
  strLinePtr = new char[strlen(tempLinePtr)+1];
  //copy the buffer to this line
  strcpy(strLinePtr, tempLinePtr);
  //put the line in the tofile
  out<<strLinePtr<<endl;
  //continue until you reach the eof
}while(!in.eof());
//close the two file streams when done with the copying
in.close();
out.close();
```

```
  //free the memory that was allocated to the buffer
  delete[] tempLinePtr;

  return;
}

//------------------------------------------------------------------------------
//function: copytobuffer(char* fromfile, char* tobuffer)
//return value: char*, pointer to the buffer with the file contents
//parameters: char* fromfile, char* tobuffer
//purpose: copies line by line the contents of fromfile to tobuffer
//      The do while executed at least once so it can read the eof of an
//      empty file
//------------------------------------------------------------------------------
//
char* copytobuffer(char* fromfile, char* tobuffer,int key_index, int key)
{

  char* strLinePtr;
  char* tempLinePtr;


 //THE OBJECT sout DYNAMICALLY ALLOCATES MEMORY FOR BUFFERING
  ostrstream sout;

  //copy the contents of fromfile to tobuffer
  //open the file appropriately
  fstream in(fromfile, ios::in);

  do{//read each line
    //allocate temporary memory for the contents of each line
    tempLinePtr = new char[MAX_LINE_SIZE];

    in.getline(tempLinePtr,MAX_LINE_SIZE,'\n');
    //allocate exactly so many memory as needed tor this line
    strLinePtr = new char[strlen(tempLinePtr)+1];
    //copy the buffer to this line

    strcpy(strLinePtr, tempLinePtr);
    //deallocate memory
    delete [] tempLinePtr;

    if (in.eof()){
      //deallocate memory and break
       delete[] strLinePtr;
       break;
    }
// Breakpoint (if you modify the driver to create more than 10 packets)
    //put the line in the ostrstream object
     sout << strLinePtr << endl;
//deallocate the memory because the object dynamically
     // allocates memory itself
     delete[] strLinePtr;
//continue until you reach the eof
  }while(!in.eof());

  //close the  file stream when done with the copying
  in.close();

  sout<<key_index<<key<<ends;

  //point char* tobuffer to the contents of the ostrstream object
  char*tempbuf=sout.str();
  tobuffer = new char [strlen(tempbuf)+1];
  //tobuffer = sout.str();
  strcpy(tobuffer,tempbuf);
  return tobuffer;
}
//end fileutils.cpp
```

```
#ifndef __global_H__
#define __global_H__
/* GLOBAL.H - RSAREF types and constants
 */

/* PROTOTYPES should be set to one if and only if the compiler supports
  function argument prototyping.
The following makes PROTOTYPES default to 0 if it has not already
  been defined with C compiler flags.
 */


/*
#ifndef PROTOTYPES
#define PROTOTYPES 0
#endif
*/

/* POINTER defines a generic pointer type */
typedef unsigned char *POINTER;

/* UINT2 defines a two byte word */
typedef unsigned short int UINT2;

/* UINT4 defines a four byte word */
#if defined(__alpha)
typedef unsigned int UINT4;
#else
typedef unsigned long int UINT4;
#endif

/* PROTO_LIST is defined depending on how PROTOTYPES is defined above.
If using PROTOTYPES, then PROTO_LIST returns the list, otherwise it
  returns an empty list.
 */
```

```
/*#if PROTOTYPES
#define PROTO_LIST(list) list
#else
#define PROTO_LIST(list) ()
#endif
*/

#endif
```

```
#ifndef __hashutils_H__
#define __hashutils_H__

//***************************************************************************
//***************************************************************************
// File : hashutils.h
// Author : Ioannis Kondoulis
//
// Operating Enviroment: Windows NT 4.0
// compiler: Borland C++ for Windows, ver. 5.02
// Date: 17 Sep 1998
// Description: This is an interface for the MD5 functions that can be
//          that can be found in MD5.c file. The code here is not all mine.
//          I wrote the two comparison functions. The rest I modified from a
//          driver program that was written by Andy Brown and I downloaded
//          from the web.
//
// Assumptions:none
//
// Warnings: n/a
//
//***************************************************************************
//


#include "md5.h"
#include "global.h"

#define TEST_BLOCK_LEN 1000
#define TEST_BLOCK_COUNT 1000
#define MD 5
#define NULL 0

typedef int boolean;
```

```
#define true 1
#define false 0

/*prototypes*/
 unsigned char* MD5File (char* );
 unsigned char* MD5String (char*);
 void MD5TestString (char*);
 void MD5Print (unsigned char*);
 boolean compareDigest (unsigned char*, unsigned char*);
 boolean compare2Digest (unsigned char*, unsigned char*);


 #endif
```

```
//******************************************************************************
//******************************************************************************
// File : hashutils.c
// Author : Ioannis Kondoulis
//// Operating Enviroment: Windows NT 4.0
// compiler: Borland C++ for Windows, ver. 5.02
// Date: 17 Sep 1998
// Description: This is an interface for the MD5 functions that can be
//          that can be found in MD5.c file. The code here is not all mine.
//          I wrote the two comparison functions. The rest I modified from a
//          driver program that was written by Andy Brown and I downloaded
//          from the web.
//
// Assumptions:none
//
// Warnings: the 5 warnings that you will get are mainly due to the use
//        of C code that makes the compiler complain.
//        They do not affect the program execution.
//
//*****************************************************************************
//
#include "hashutils.h"
#include <stdio.h>
#include <string.h>

 unsigned char* MD5File (filename)
char *filename;
{
  FILE *file;
  MD5_CTX context;
  int len;
  unsigned char buffer[1024], digest[16];
  unsigned char* new_digest = malloc(sizeof(digest));
  int i;
  if ((file = fopen (filename, "rb")) == NULL)
    printf ("%s can't be opened\n", filename);


  else {
    MD5Init (&context);
    while (len = fread (buffer, 1, 1024, file))
      MD5Update (&context, buffer, len);
    MD5Final (digest, &context);


    fclose (file);


    printf ("MD%d (%s) = ", MD, filename);
    MD5Print (digest);
    printf ("\n");
  }
  for (i=0; i<=16; i++)
   new_digest[i] = digest[i];
   /*CAUTION: uncomment for debugging purposes only
   printf ("\n");
   MD5Print (new_digest);
   printf ("\n");
   */
   return new_digest;
}//end MD5File

/* Digests the standard input and prints the result.
 */
 /* Prints a message digest in hexadecimal.
 */
void MD5Print (digest)
unsigned char digest[16];
{
  unsigned int i;

  for (i = 0; i < 16; i++)
```

```
    printf ("%02x", digest[i]);

}//end MD5Print

boolean compareDigest (unsigned char* s1, unsigned char* s2)
{
  int i;
  boolean same = true;
  for (i = 0; i<16;i++){
    if ((s1[i])^(s2[i])){
      same = false;
      break;
    }
  }
  return same;
}

boolean compare2Digest (unsigned char* s1, unsigned char* s2)
{
  int i;
  boolean same = true;

  /*convert each Message Digest to four 4-byte integers
  CAUTION: THIS PART MAY NOT BE PORTABLE
  */

  /*ff all the differeces are equal to 0 then the digests are the same
  */

  for (i=0; i<16;){
    if (s1[i]-s2[i]){
      same = false;
      break;
    }
    i+=4;
```

```
   }

  return same;
}
/* Digests a string and prints the result.
 */
unsigned char* MD5String (string)
char *string;
{
  MD5_CTX context;
  unsigned char digest[16];
  unsigned int len = strlen (string);
  unsigned char* new_digest = malloc(sizeof(digest));
  int i;

  MD5Init (&context);
  MD5Update (&context, string, len);
  MD5Final (digest, &context);

  /*CAUTION: uncomment for debugging purposes only
  printf ("MD%d (\"%s\") = ", MD, string);
  MD5Print (digest);
  */
  for (i=0; i<=16; i++)
  new_digest[i] = digest[i];
  /*CAUTION: uncomment for debugging purposes only
  printf ("\n");
  MD5Print (new_digest);
  printf ("\n");
  */
  return new_digest;
}//end MD5String

/* Digests a string, it is used only for testing the performance.
 */
```

```
void MD5TestString (string)
char *string;
{
  MD5_CTX context;
  unsigned char digest[16];
  unsigned int len = strlen (string);

  MD5Init (&context);
  MD5Update (&context, string, len);
  MD5Final (digest, &context);

}//end MD5TestString
```

```
#ifndef __host_key_class_H__
#define __host_key_class_H__

//********************************************************************************
//********************************************************************************
// File : host_key_class.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:This is the class that contains the host_id and the host_key
//         The objects are used by the kerberos as entries in a database.
//         They are created by kerberos whenever a host requests to
//         register with kerberos.
//         The registration function will be simulated for this demo.
//         The key exchange process is not of interest to us (we have to
//         assume that somehow the kerberos and the host are aware of the
//         host's key) therefore we will  hardcode (we will assign) the key
//         for a particular host at a certain point during the simulation
//         and we are also inform the kerberos for the hosts key (the
//         programmer takes the place of secure key distributor)
//         The necessary functions to access the data are also
//         provided.
//
// Assumptions: the overloaded operators <<, >> are used
//          only by authorized users
//:
//
//********************************************************************************
//
#include <iostream>

class host_key_class {
  //friend classes
  friend class kerberosclass;
  friend class host_key_vector;

  //overloading i/o operators
  friend ostream& operator<<(ostream& out, const host_key_class& MY_HK);
  friend istream& operator>>(istream& in, host_key_class& my_hk );
public:
  //constructor
  host_key_class();

  //another constructor
  host_key_class(int host, char* key);

  //destructor
  ~host_key_class();

private:
  //DATA
  //for each host we assume there exists a key known to kerberos
  //and to the host only
  int   host_id;
  char* host_key;

  //service to kerberos
  int host_id_is() const{return host_id;}
  char* key_is()const {return host_key;}

};//end host_key_class

#endif
//end host_key_class.h file
```

```
#ifndef __hashutils_H__
#define __hashutils_H__

//********************************************************************************
//********************************************************************************
// File : hashutils.h
// Author : Ioannis Kondoulis
//
// Operating Enviroment: Windows NT 4.0
// compiler: Borland C++ for Windows, ver. 5.02
// Date: 17 Sep 1998
// Description: This is an interface for the MD5 functions that can be
//          that can be found in MD5.c file. The code here is not all mine.
//          I wrote the two comparison functions. The rest I modified from a
//          driver program that was written by Andy Brown and I downloaded
//          from the web.
//
// Assumptions:none
//
// Warnings: n/a
//
//********************************************************************************
//
#include "md5.h"
#include "global.h"

#define TEST_BLOCK_LEN 1000
#define TEST_BLOCK_COUNT 1000
#define MD 5
#define NULL 0

typedef int boolean;

#define true 1
#define false 0
```

```
/*prototypes*/
 unsigned char* MD5File (char* );
 unsigned char* MD5String (char*);
 void MD5TestString (char*);
 void MD5Print (unsigned char*);
 boolean compareDigest (unsigned char*, unsigned char*);
 boolean compare2Digest (unsigned char*, unsigned char*);


#endif
```

```
#ifndef __host_key_vector_H__
#define __host_key_vector_H__

//***************************************************************************
//***************************************************************************
// File : host_key_vector.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:This is the database of host_key_class objects. This class is
//          used by the kerberos module for keeping track of the hosts and
//          their keys.
//           The STL::vector data structure was chosen to contain
//          host_key_class objects. The vector dynamically extends to
//          accomodate each additional entry and automatically deallocates
//          memory when we remove an object.
//          The choice was made for simplicity for the demo purposes
//          In a more elaborate implementation a more sophisticated database
//          might be used.
//          The member functions are private so only the kerberos can get
//          their services.
// Assumptions:for each host we assume there exists a key known to kerberos
//          and to the host only
//
// Warnings
//
//***************************************************************************
//
#include <vector>
#include <stdlib>
#include "host_key_class.h"

using namespace std;
```

```
//a vector is used for the host_key_class object database
typedef vector<host_key_class> host_database;

class host_key_vector {
  //friend class
  friend class kerberosclass;
  //overloading i/o operators
  friend ostream& operator<<(ostream& out, const host_key_vector& MY_HKV);
  //does not make sense to input to a vector so no  >> is overloaded

public:
  //constructor
  host_key_vector();

  //destructor
  ~host_key_vector();

private:
  //data
  host_database hosts_keys;

  //member functions for friends
  //services to the kerberos module
  void add_host   (int host, char* key);
  void remove_host(int host);
  bool is_host_registered (int host);

};//end host_key_vector class

#endif

//end host_key_vector.h file
```

```
//******************************************************************************
//******************************************************************************
// File : host_key_vector.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description: the same as in host_key_vector.h
// Assumptions: none
//
//
//
//******************************************************************************
//
#include "host_key_vector.h"

//-------------------------------------------------------------------------------
//function: default constructor
//return value: a host_key_vector
//parameters: none
//purpose: creates a host_key_vector object that is empty
//-------------------------------------------------------------------------------
//
host_key_vector::host_key_vector(){};

//-------------------------------------------------------------------------------
//function: destructor
//return value: none
//parameters:  none
//purpose: destroys "this" host_key_vector object
//-------------------------------------------------------------------------------
//
host_key_vector::~host_key_vector(){};
```

```
//----------------------------------------------------------------------------
//function: add_host(int host,char* key)
//return value: none
//parameters: int host, char* key
//purpose:  This function is a service to the kerberos object.
//        first we create a host_key_class object that will be the
//        new entry to the database (vector),then we use the STL::vector
//        member function "push_back(class T)" that adds this new entry to the vector
//
//----------------------------------------------------------------------------
//
void  host_key_vector::add_host(int host,char* key)
{
  //create new object
  host_key_class new_host(host,key);
  hosts_keys.push_back(new_host);
  return;
}//end add host(int host,char* key)

//----------------------------------------------------------------------------
//function: remove_host(int host)
//return value: none
//parameters: int host
//purpose:  This function is a service to the kerberos object.
//        here only the host (host_id) is required in order to find the
//        corresponding entry (host_key object) and remove it. A vector
//        iterator is used to traverse the vector object, find the entry and
//        remove it. Multiple entries are removed also. We use the STL::vector
//        member function "erase(iterator i)" that removes the object pointed
//        by i from the vector object.
//
//----------------------------------------------------------------------------
//
void host_key_vector::remove_host(int host)
{
  //traverse the vector
  for (host_database::iterator i= hosts_keys.begin();
    i!=hosts_keys.end(); i++){
    //if you find the host
    if (i->host_id_is() == host)
      hosts_keys.erase(i);
    }
  return;
}//end remove_host(int host)

//---------------------------------------------------------------------------
//function: is_host_registered(int host)
//return value: boolean
//parameters: int host
//purpose:  This function is a service to the kerberos object.
//        here only the host (host_id) is required in order to find if
//        a corresponding entry (host_key object) exists.A vector
//        iterator is used to traverse the vector object.Only one appearance is
//        enough to make the return value "true". True means that there exists
//        a host_key entry in the kerberos' database
//---------------------------------------------------------------------------
//
bool host_key_vector::is_host_registered(int host)
{
  //local variable
  bool host_registered = false;
  //traverse the vector
  for (host_database::iterator i= hosts_keys.begin();
    i!=hosts_keys.end(); i++) {//safer than int i etc
    //if you find the host_id
    if (i->host_id_is() == host){
      host_registered = true;
      //once is enough
      break;
    }//end if
  }//endfor
 return host_registered;
 }//end is_host_registered(int host)

//-----------------------------------------------------------------------------
//function: ostream& operator<<(ostream& out, const host_key_vector& MY_HKV)
//return value:ostream&
//parameters:  ostream  out,host_key_vector& MY_HKV
//purpose: a user defined class has to define how << behaves.
//        In our case if the host_key_vector  object passed in by ref
//        does not contain anything then we only inform the user that
//        the object is empty
//        otherwise we display all the host_id and the host_key pairs
//        contained in the vector.
//-----------------------------------------------------------------------------
//
ostream& operator<<(ostream& out, const host_key_vector& MY_HKV)
{
  if (MY_HKV.hosts_keys.size()<= 0 ){
    //the vector is empty
    out << "<host_key_database is empty>"<<endl;
  }
  else{//traverse the vector
    for (host_database::iterator i = const_cast<host_key_class *>
                                (MY_HKV.hosts_keys.begin());
      i!=MY_HKV.hosts_keys.end(); i++){
      //if you find the user entry
      out<<(*i)<<endl;
    }//endfor

  }//end if
  return out;
}//end operator<<(ostream& out, const host_key_vector& MY_HKV)
```

```
//end host_key_vector.cpp file
```

```
#ifndef __host_table_class_H__
#define __host_table_class_H__

//********************************************************************************
//********************************************************************************
// File : host_table_class.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:This is the class that contains the host_id and a pointer to the
//          tableclass object that was created by the nacclass object for
//          that host
//          The objects are used by the nacclass object (nac) as entries
//          in a database(vector) so that the nac can keep track of what
//          table that host has been allocated for further usage.
//          They are created by nacclass whenever a host requests a table
//          after the ticket of the host has been checked and has been found
//          correct.
//          The necessary functions to access the data are also
//          provided.
//
// Assumptions: none
//
//
//********************************************************************************
//
#include "tableclass.h"

class host_table_class{
  //friend classes
  friend class nacclass;
  friend class host_table_vector;
public:
  //default constructor
  host_table_class();

  //another constructor
  explicit host_table_class(int host_id,
  // TO DO CHECK THE CONSTNESS OF THE POINTER
          tableclass * const TABLEPTR);
  //destructor
  ~host_table_class();

private:
  //data
  int host_id;
  tableclass * tableptr;

  //member functions for friends
  int host_id_is()const{return host_id;}
  tableclass* table_is()const {return tableptr;}
};//end host_table_class

#endif

//end host_table_class.h file
```

```
//****************************************************************************
//****************************************************************************
// File : host_table_class.cpp
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:same as in host_table_class.h
//
// Assumptions:same as in host_table_class.h
//
//
//****************************************************************************
//
#include "host_table_class.h"

//---------------------------------------------------------------------------
//function: default constructor
//return value: a host_table_class object
//parameters: none
//purpose: the initial values are entered in order to be able to check the
//    existance of a host_id and a tableptr.The host_table_class object
//    is initialized with host_id:-1 and no tableptr(by default it is nulled).
//---------------------------------------------------------------------------
//
host_table_class::host_table_class():host_id(-1){}

//---------------------------------------------------------------------------
//function: another constructor
//return value: a host_table_class object
//parameters: int host_id, tableclass* TABLEPTR
//purpose: the host_id and the TABLEPTR are entered to the created
//    object. The object is going to be added to the host_table_vector database
//    object for further usage by the nac.
```

```
//---------------------------------------------------------------------------
//
host_table_class::host_table_class(int host_id,
                           tableclass * const TABLEPTR):
host_id(host_id),tableptr(TABLEPTR){}

//---------------------------------------------------------------------------
//function: destructor
//return value: none
//parameters:  none
//purpose: destroys "this" host_table_class object
//---------------------------------------------------------------------------
//
host_table_class::~host_table_class(){}

//end host_table_class.cpp file
```

```
#ifndef __host_table_vector_H__
#define __host_table_vector_H__

//****************************************************************************
//****************************************************************************
// File : host_table_vector.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:This is the database of host_table_class objects. This class is
//          used by the nac module for keeping track of the hosts and the
//          ki_key tables they are assigned
//           The STL::vector data structure was chosen to contain
//          host_table_class objects. The vector dynamically extends to
//          accomodate each additional entry and automatically deallocates
//          memory when we remove an object.
//          The choice was made for simplicity for the demo purposes
//          In a more elaborate implementation a more sophisticated database
//          might be used.
//          The member functions are private so only the nac can get
//          their services.
// Assumptions: none
//
// Warnings
//
//****************************************************************************
//
#include <vector>
#include <stdlib>
#include "host_table_class.h"
#include "tableclass.h"
#include <iostream>
```

```
using namespace std;

//a vector is used for the host_table_class object database
typedef vector<host_table_class>
                    host_table_database;

class host_table_vector {
  //friend class
  friend class nacclass;

public:
  //constructor
  host_table_vector();

  //destructor
  ~host_table_vector();

  //for debugging purposes
  void print_vector();

int table_size_is()const{return hosts_tables.size();}
private:
  //data
  host_table_database hosts_tables;

  //member functions for friends
  //services to the nac module
  void add_host_table(host_table_class host_table);
  void remove_host_table(int host);
  bool is_host_authorized(int host);
  int  key_for_host_ki(int host, int ki);
};//end host_key_vector class
#endif

//end host_table_vector.h file
```

```
//**************************************************************************************
//**************************************************************************************
// File : host_table_vector.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:the same as in host_table_vector.h
// Assumptions: none
//
// Warnings
//
//**************************************************************************************
//
#include "host_table_vector.h"

//---------------------------------------------------------------------------
//function: default constructor
//return value: a host_table_vector
//parameters: none
//purpose: creates a host_table_vector object that is empty
//---------------------------------------------------------------------------
//
host_table_vector::host_table_vector(){}

//---------------------------------------------------------------------------
//function: destructor
//return value: none
//parameters:  none
//purpose: destroys "this" host_table_vector object
//---------------------------------------------------------------------------
//
host_table_vector::~host_table_vector(){}
```

```
//---------------------------------------------------------------------------
//function: add_host(host_table_class host_table)
//return value: none
//parameters: host_table_class host_table
//purpose:  This function is a service to the nacclass object.
//        the host_table_class object that will be the
//        new entry to the database (vector),is passed in by value.
//        Then we use the STL::vector member function "push_back(class T)"
//        that adds this new entry to the vector
//
//---------------------------------------------------------------------------
//
void  host_table_vector::add_host_table(host_table_class host_table)
{
  hosts_tables.push_back(host_table);
  return;
}//end add_host_table(host_table_class host_table)

//---------------------------------------------------------------------------
//function: remove_host(int host)
//return value: none
//parameters: int host
//purpose:  This function is a service to the nacclass object.
//       here only the host (host_id) is required in order to find the
//       corresponding entry (host_key object) and remove it. A vector
//       iterator is used to traverse the vector object, find the entry and
//       remove it. Multiple entries are removed also. We use the STL::vector
//       member function "erase(iterator i)" that removes the object pointed
//       by i from the vector object.
//
//---------------------------------------------------------------------------
//
void host_table_vector::remove_host_table(int host)
{
  //traverse the vector
  for (host_table_database::iterator i= hosts_tables.begin();
    i!=hosts_tables.end(); i++){
    //if you find the host
    if (i->host_id_is() == host){
      hosts_tables.erase(i);
    }//endif
  }//end for
  return;
}//end remove_host_table(int host)

//---------------------------------------------------------------------------
//function: is_host_authorized(int host)
//return value: boolean
//parameters: int host
//purpose:  This function is a service to the nac object.
//        here only the host (host_id) is required in order to find if
//        a corresponding entry (host_key object) exists.A vector
//        iterator is used to traverse the vector object.Only one appearance is
//        enough to make the return value "true". True means that there exists
//        a host_table entry in the nac's database, therefore the host is
//        authorized to send traffic
//---------------------------------------------------------------------------
//
bool host_table_vector::is_host_authorized(int host)
{
  //local variable
  bool host_exists = false;

  //traverse the vector
  for (host_table_database::iterator i= hosts_tables.begin();
    i!=hosts_tables.end(); i++) {//safer than int i etc
    //if you find the host_id
    if (i->host_id_is() == host){
      host_exists = true;
      //once is enough
      break;
    }//endif
  }//endfor
  return host_exists;
}//end is_host_authorized(int host)

//---------------------------------------------------------------------------
//function: key_for_host_ki(int host, int ki)
//return value: int, the key
//parameters: int host, int ki
//purpose:  This function is a service to the nac object.
//        here only the host (host_id) is required in order to find if
//        a corresponding entry (host_key object) exists.A vector
//        iterator is used to traverse the vector object. The entry for host
//        is found and a handler to the tableclass object of this entry is
//        returned. The function now uses the service of the tableclass object
//        look_up_table_for_key(ki) in order to find the key value that it
//        returns.
//---------------------------------------------------------------------------
//
int host_table_vector::key_for_host_ki(int host, int ki)
{
  //local handler to the table
  tableclass *tableptr;
  //traverse the vector and look for that host entry
  for (host_table_database::iterator i= hosts_tables.begin();
    i!=hosts_tables.end(); i++){//safer than int i etc
    //alternatively IF int i had been used to traverse
    //we could use:
    //if (hosts_tables.operator[](i).host_id_is() == host){
    //     tableptr = hosts_tables.operator[](i).table_is();
    if (i->host_id_is() == host){
      //a local hanler to the table of the host
      tableptr = i ->table_is();
      break;
    }//end if
  }//end for
  //use the service provided by the tableclass to the friends
  return (tableptr -> look_up_table_for_key(ki));
}//end key_for_host_ki

//---------------------------------------------------------------------------
//function: print_vector()
//return value: none
//parameters: none
//purpose: FOR DEBUGGING PURPOSES ONLY.
//---------------------------------------------------------------------------
//
void host_table_vector::print_vector()
{
  for (host_table_database::iterator i= hosts_tables.begin();
    i!=hosts_tables.end(); i++){//safer
    cout<<"for host:"<< i->host_id_is()
       << " ki_key table is:" <<endl;
       i->table_is()->print_table();
  }//end for
  return;
}//end print_vector

//end host_table_vector.cpp file
```

```
#ifndef __host_ticket_class_H__
#define __host_ticket_class_H__

//****************************************************************************
//****************************************************************************
// File : host_ticket_class.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:This is the class that contains the host_id and the name of
//          the file that contains the ticket. The objects are created by
//          the host class and are sent to the nac for further authorization
//          (getting a tableclass)so the host can start sending traffic//
//          to the nac. The necessary functions to access the data are also
//          provided.
//
// Assumptions: The file that contains the ticket is encrypted with the
//          encryption method chosen for the protocol (IDEA in our case)
//
//
//
//****************************************************************************
//
#include <iostream>

class host_ticket_class {
         friend class nacclass;
  //overloading of operators
  friend ostream& operator<<(ostream&, const host_ticket_class&);
  friend istream& operator>>(istream&, host_ticket_class& );
public:
  //default constructor
  host_ticket_class();
//another constructor
  explicit host_ticket_class(int host_id,
                        char* ticket_file);
  //destructor
  ~host_ticket_class();

private:
  //data
  int host_id;
  char* ticket_file;

  //member functions
  //for friends
  int host_id_is()const{return(host_id);}
  char* ticket_file_is()const{return(ticket_file);}

};//end host_ticket_class

#endif

//end host_ticket_class.h file
```

```
//***************************************************************************
//***************************************************************************
// File : host_ticket_class.cpp
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:same as in host_ticket_class.h
//
// Assumptions: same as in host_ticket_class.h
//
//
//
//***************************************************************************
//
#include "host_ticket_class.h"

//----------------------------------------------------------------------------
//function: default constructor
//return value: a host_ticket_class object
//parameters: none
//purpose: the initial values are entered in order to be able to check the
//   existance of a host-id and a file name.The host_ticket_class object
//    is initialized with host_id:-1 and file name : "notexists".
//----------------------------------------------------------------------------
//
host_ticket_class::host_ticket_class ():
host_id(-1),ticket_file("notexists"){}

//----------------------------------------------------------------------------
//function: another constructor
//return value: a host_ticket_class object
//parameters: int host_id, char* ticket_file
//purpose: the host_id and the ticket_file name are entered to the created
```

```
//   object by the host. The object is going to be sent to the nac for further
//   usage.
//---------------------------------------------------------------------------
//
host_ticket_class::host_ticket_class (int host_id, char* ticket_file):
host_id(host_id),ticket_file(ticket_file){}

//---------------------------------------------------------------------------
//function: destructor
//return value: none
//parameters:  none
//purpose: destroys "this" host_ticket_class object
//---------------------------------------------------------------------------
//
host_ticket_class::~host_ticket_class(){}

//---------------------------------------------------------------------------
//function: ostream& operator<<(ostream& out, const host_ticket_class& MY_HT)
//return value:ostream&
//parameters:  ostream  out,host_ticket_class& MY_HT
//purpose: a user defined class has to define how << behaves.
//         In our case if the host_ticket_class object passed in by ref
//         has not been initialized (host_id and ticket_file  are
//         not valid) then we only inform the user that the object does not
//         exist, otherwise we display the host_id and the ticket file name.
//
//remark: even one unitialized value would lead to an invalid object
//---------------------------------------------------------------------------
//
ostream& operator<<(ostream& out, const host_ticket_class& MY_HT)
{
  if ((MY_HT.host_id <0 )&&
          (MY_HT.ticket_file == "notexists")){
    //for some reason the key_ticket object is not there
    out << "<host_ticket object does not exist>"<<endl;
  }
  else if (MY_HT.host_id < 0 ){
    //for some reason the key_ticket object's session_key is not there
    out<<"<host_ticket object.host_id does not exist> "<< endl;
  }
  else if (MY_HT.ticket_file=="notexists"){
    //for some reason the key_ticket object's ticket_file is not there
    out<<"<host_ticket object.ticket_file does not exist> "<< endl;
  }
  else{//everything exists
            out << MY_HT.host_id<<endl;
            out << MY_HT.ticket_file<<endl;
  }//end if
  return out;
}//end operator<<(ostream& out, const host_ticket_class& MY_HT)

//---------------------------------------------------------------------------
//function: istream& operator>>(istream& in, host_ticket_class& my_ht )
//return value:istream&
//parameters:  istream& in,host_ticket_class my_ht
//purpose: a user defined class has to define how >> behaves.
//       Here we input the two values to the host_ticket_class object
//        on the RHS
//---------------------------------------------------------------------------
//
istream& operator>>(istream& in, host_ticket_class& my_ht )
{
          in >>my_ht.host_id;
  in >>my_ht.ticket_file;
  return in;
}//end operator>>(istream& in, host_ticket_class& my_ht )

//end file host_ticket_class.cpp
```

```
#ifndef __hostclass_H__
#define __hostclass_H__
//*****************************************************************************
//*****************************************************************************
// File : hostclass.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
 // Description: the hostclass simulates the functionality of a host that wants
//          to get authorization from kerberos, obtain a ticket and a session
//          key, contact nac, obtain a key-table and start sending
//          ip_packets formatted in accordance we the protocol we propose.
//           The hostclass object data consist of a host_id, host_key and a
//          user_id. So we consider the user is part of this entity (object).
//          The functionality is devided to three parts: the kerberos part
//          the nac part and ip_packet part.
//
// Assumptions: none
//
// Warnings: n/a
//
//*****************************************************************************
//
#include  "ip_packetclass.h"
#include  "tableclass.h"
#include  "ticketclass.h"
#include  "host_ticket_class.h"
#include  "key_ticket_class.h"
#include  "kerberosclass.h"
#include  "constants.h"
#include  "fileutils.h"
#include <iostream>
#include <fstream>
```

```
#include <string>

//names of the files are specified in the constants.h
//we use files to deposit and retrieve information easily
extern const char* MESSAGE_FILE ;
extern const char* HOST_TICKET_FILE ;

class hostclass{

public:
//constructors destructors
//user defined constructor
  explicit hostclass(int host_id,
              char* host_key,
              char* user_id);

  //destructor
  ~hostclass();

  //member functions
  //--------------------------------------------------------------------------
  //functionality with kerberos
  //--------------------------------------------------------------------------
  //the host requests authentication from the kerberos module
  //passing the host_id, host_key and user_id to the appropriate kerb function
  //postcondition: now the host has the name of the file that contains
  //        the session_key and the ticket. The file and the ticket should be
  //        encrypted. We do not use encryption in this simulation.
  //--------------------------------------------------------------------------
  //
        bool request_authentication(kerberosclass &kerberos);

  //--------------------------------------------------------------------------
  //functionality with nac
  //--------------------------------------------------------------------------
  //--------------------------------------------------------------------------
  //the file with the info for the nac: host_id and ticket is created and
  //the function returns the file name
  //postcondition : HOST_TICKET_FILE contains the information intended for NAC.
  //--------------------------------------------------------------------------
  //
  char* send_ticket();

  //--------------------------------------------------------------------------
  //NAC prepares a file containing a key-table, this filename is the argument
  //of this function
  //postcondition: the host extracts the table of ki_keys from the file
  //          and the table object in the host is instantiated.
  //--------------------------------------------------------------------------
  //
  void hostclass::get_table(const char* TABLE_FILE_NAME);

  //--------------------------------------------------------------------------
  //functionality with ip_packet
  //--------------------------------------------------------------------------
  //the message_file name is passed in, the  host prepares an ip_packetclass
  //object and returns it
  //postcondition: a new ip_packetclass object is created.
  //--------------------------------------------------------------------------
  //
  ip_packetclass create_message(char* message_file);

private:
  //data members
  //basic
  int host_id;
  char* host_key;
   char* user_id;

  //additional
  tableclass ki_key_table;
  char* key_ticket_file;
  char* ticket_file;
  char* session_key;

  //--------------------------------------------------------------------------
  //auxilliary function
  //postcondition: data members session_key and ticket file are defined.
  //--------------------------------------------------------------------------
  void get_session_key();

};//end hostclass

#endif
//end file hostclass.h
```

```
//****************************************************************************
//****************************************************************************
// File : hostclass.cpp
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:same as in hostclass.h
//
// Assumptions
//
// Warnings
//
//****************************************************************************
//
#include "hostclass.h"
extern "C"{
#include "hashutils.h"
#include "md5.h"
}
#include <time>

//--------------------------------------------------------------------------
//function:  constructor
//return value: a hostclass object
//parameters: int host_id,char* host_key,char* user_id
//purpose: creates a hostclass object with the values of the parameters
//                              passed in.
//--------------------------------------------------------------------------
//
hostclass::hostclass(int host_id,char* host_key,char* user_id):
        host_id (host_id),
        host_key(host_key),
        user_id (user_id){}
```

```
//---------------------------------------------------------------------------
//function: destructor
//return value: none
//parameters:  none
//purpose: destroys "this" hostclass object.
//---------------------------------------------------------------------------
//
hostclass::~hostclass(){}


//member functions
//---------------------------------------------------------------------------
//functionality with kerberos
//---------------------------------------------------------------------------
//
//---------------------------------------------------------------------------
//function: request_authentication(kerberosclass &kerberos)
//return value: bool
//parameters: kerberosclass &kerberos
//purpose:  the host requests authentication from the kerberos module
//          passing the host_id, host_key and user_id to the appropriate
//          kerberos function. Returns true if an entry exists in the
//          kerberos databases for both the user and the host.
//postcondition: the host obtains the file that contains the session_key and
//          the ticketfile name and extracts the session key using the auxilliary
//          function "get_session_key".
//
//remarks: here we assume that the driver program will allow
//          for a reference to a kerberos object
//---------------------------------------------------------------------------
//
bool hostclass::request_authentication(kerberosclass &kerberos)
{
  //at the beginning
  bool authenticated = false;
  //use kerberos functionality
  if(kerberos.request_permission(user_id,host_id)){
  //get the encrypted file with the sessionkey and the ticketfile name
    key_ticket_file =
          const_cast<char*>(kerberos.send_key_ticket(host_id));
    authenticated = true;
  }//endif
  //so the host can proceed and extract the session key
  get_session_key();
  return authenticated;
}//end request_authentication(kerberosclass &kerberos)

//---------------------------------------------------------------------------
//function: get_session_key()
//return value: none
//parameters: none
//purpose: a utility function that reads from the file that kerberos
//          created the session_key and the encrypted ticketfile name
//---------------------------------------------------------------------------
//
void hostclass::get_session_key()
{
          //a local variable to read in from the file
          key_ticket_class key_ticket;

          //first need to decrypt the file,use host's key(NOT IMPLEMENTED)
          //decrypt(key_ticket_file, "hostkey");

          //open the file to get a value (for input)
          fstream in (key_ticket_file, ios::in);
  //use overloaded >>
          in >> key_ticket;

          //instantiate this hostclass object's data members
          session_key = key_ticket.session_key_is();
          ticket_file = key_ticket.ticket_file_is();
```

```
        in.close();
        return;
}//get_session_key()




//---------------------------------------------------------------------------
//functionality with nac
//---------------------------------------------------------------------------
//
//---------------------------------------------------------------------------
//function: send_ticket()
//return value: none
//parameters: none
//purpose:the host creates a host_ticket_class object (contains the host_id
//                  and the ticketfile name) opens a file and dumps the object
//                  in the file. The function returns the name of this file.
//---------------------------------------------------------------------------
//
char* hostclass::send_ticket()
{
        //the host opens a file and puts inside the host_id
        //and the name of the ticketfile
        host_ticket_class host_ticket(host_id,ticket_file);

        //open the file for output
        fstream out(HOST_TICKET_FILE, ios::out);
        //use overloaded <<
        out<<host_ticket;
        //for safety
        out.close();

        return const_cast<char*>(HOST_TICKET_FILE);
}//end send_ticket
```

```
//--------------------------------------------------------------------------
//function: get_table(const char* TABLE_FILE_NAME)
//return value: none
//parameters: const char* TABLE_FILE_NAME
//purpose:the host opens a file that was created by the nac and contains the
//                          ki- key table.
//postcondition: the tableclass object data member is instantiated.
//--------------------------------------------------------------------------
//
void hostclass::get_table(const char* TABLE_FILE_NAME)
{
        //the file should be decrypted using the session key (NOT IMPLEMENTED)
        //decrypt(TABLE_FILE_NAME,"sesskey")

        //open the file input
        fstream in(TABLE_FILE_NAME, ios::in);
        //read in the table, use overloade >>
        in >> ki_key_table;

        return;
}//end get_table(const char* TABLE_FILE_NAME)

//--------------------------------------------------------------------------
//functionality with ip_packet
//--------------------------------------------------------------------------
//
//--------------------------------------------------------------------------
//function: create_message(char* message_file)
//return value: none
//parameters: char* message_file
//purpose:the host prepares an ip_packetclass object. First, we get the new
//                          ki and the next key from the key-table.Then we copy the message file
//                          to buffer, append the key and hash the whole file
//                          thus creating the MAC.
//Postcondition: The new ip_packetclass object is created and returned.
```

```
//
//-----------------------------------------------------------------------------
//
ip_packetclass hostclass::create_message(char* message_file)
{
  //a pointer to a buffer to be used for MD5(buffer)

  char* message_buffer;
  char* temp_buffer= new char[MAX_MESSAGE_SIZE];
  static int new_ki;
  static int indicator=0;
  static time_t start_time;
  time_t current_time;

  if (indicator==0){//its the first time
    start_time=time(0);
    new_ki = ki_key_table.get_next_ki();
    indicator=1;
  }

  current_time=time(0);

  //the next ki and key are being requested for the message
  //use table functionality
  if ((current_time-start_time)>T_ZERO){
    cout<<"**The old key's cryptoperiod has expired.A new key is valid**\n"
       <<endl;
    new_ki = ki_key_table.get_next_ki();
    //reset the timer
    start_time=time(0);
  }
  int next_key= ki_key_table.get_next_key(new_ki);

  //the message_file contents are put in a buffer and
  //the key is appended to it
```

```
    //this is done in order to keep the message_file clean
    temp_buffer = copytobuffer(message_file,temp_buffer,new_ki,next_key);

    message_buffer = new char [strlen(temp_buffer)+1];
    strcpy (message_buffer, temp_buffer);
    delete[] temp_buffer;

    // md5 is CALLED FOR THE BUFFER
    unsigned char* mac = MD5String (message_buffer);

    //create a new ip packet
            ip_packetclass * messageptr = new ip_packetclass(host_id,
            message_file,new_ki,mac);
            return *messageptr;
}//end create_message



//end file hostclass.cpp
```

```
#ifndef __ip_packetclass_H__
#define __ip_packetclass_H__

//***************************************************************************
//***************************************************************************
// File : ip_packetclass.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description: the ip_packetclass objects (ip_packets) are used by the
//          hostclass and the nacclass.
//          The hostclass prepares them and the nacclass
//          authenticates them. The information contained in each ip_packet
//          (data) is the host_id of the host that prepared the packet,
//          the ki of the packet, the filename of the message data,and
//          the mac(message authentication code) the hash value of the
//          message+key.
//
// Assumptions
//
// Warnings
//
//***************************************************************************
//
#include <iostream>

class ip_packetclass{
  //friend class
          friend class nacclass;
public:
          //constractors destructors
  //default constractor
  ip_packetclass();
  //another constructor
  explicit ip_packetclass(int id,char* data_file,int ki,unsigned char* mac);

  //copy constructor
  //NOT NECESSARY FOR THE MOMENT
  //ip_packetclass (const ip_packetclass &);

  //destructor
  ~ip_packetclass();

private:
  //data
  int ki;
  int host_id;
  //the filename that contains the message
  char *filename;
  //128 bits
  unsigned char* mac;

  //member functions for friends
          unsigned char* mac_is()const{return mac;}
  int     ki_is()const{return ki;}
  char* filename_is()const{return filename;}
          int         host_is()const{return host_id;}


};//end ip_packet class

#endif
//end file ip_packetclass.h
```

```
//****************************************************************************
//****************************************************************************
// File : ip_packetclass.cpp
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description: same as in the header file
//
// Assumptions: none
//
//
//
//****************************************************************************
#include "ip_packetclass.h"


//--------------------------------------------------------------------------
//function: default constructor
//return value: an ip_packetclass object
//parameters: none
//purpose: the initial values are entered in order to be able to check the
//existance of the packet
//--------------------------------------------------------------------------
//
ip_packetclass::ip_packetclass():host_id(-1),
          filename("no_file_exists"),
                    ki(-1),
                    mac('\0'){}

//--------------------------------------------------------------------------
//function: another constructor
//return value: a ip_packetclass object
//parameters: int id (host_id), char* data file, int ki, char* mac
```

```
//purpose: an ip_packetclass object with initial values
//          the parameters passed has been created by a hostclass object
//---------------------------------------------------------------------------
//
ip_packetclass::ip_packetclass(int id, char* data_file,
            int ki,unsigned char* mac):
filename(data_file), ki(ki), host_id (id), mac (mac){}

//---------------------------------------------------------------------------
//function: destructor
//return value: none
//parameters:  none
//purpose: destroys "this" ip_packetclass object
//---------------------------------------------------------------------------
//
ip_packetclass::~ip_packetclass(){}

//end file ip_packetclass.cpp
```

```
//*****************************************************************************
//*****************************************************************************
// File : ip_packetclass.cpp
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description: same as in the header file
//
// Assumptions: none
//
//
//
//*****************************************************************************
#include "ip_packetclass.h"

//----------------------------------------------------------------------------
//function: default constructor
//return value: an ip_packetclass object
//parameters: none
//purpose: the initial values are entered in order to be able to check the
//existance of the packet
//----------------------------------------------------------------------------
//
ip_packetclass::ip_packetclass():host_id(-1),
        filename("no_file_exists"),
                ki(-1),
                mac('\0'){}

//----------------------------------------------------------------------------
//function: another constructor
//return value: a ip_packetclass object
//parameters: int id (host_id), char* data file, int ki, char* mac
//purpose:  an ip_packetclass object with initial values
```

```
//          the parameters passed has been created by a hostclass object
//---------------------------------------------------------------------------
//
ip_packetclass::ip_packetclass(int id, char* data_file,
            int ki,unsigned char* mac):
filename(data_file), ki(ki), host_id (id), mac (mac){}

//---------------------------------------------------------------------------
//function: destructor
//return value: none
//parameters:  none
//purpose: destroys "this" ip_packetclass object
//---------------------------------------------------------------------------
//
ip_packetclass::~ip_packetclass(){}

//end file ip_packetclass.cpp
```

```
//******************************************************************************
//******************************************************************************
// File : kerberosclass.cpp
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description: the same as in kerberosclass.h file
//
// Assumptions
//
// Warnings: they are inherited from the STL vector. they do not cause
//          any problem.
//
//******************************************************************************
#include "kerberosclass.h"

//---------------------------------------------------------------------------
//function: default constructor
//return value: a kerberosclass object
//parameters: none
//purpose: creates a kerberosclass object with two empty databases
//         one for users and one for host_key_class objects
//---------------------------------------------------------------------------
//
kerberosclass::kerberosclass(){}

//---------------------------------------------------------------------------
//function: destructor
//return value: none
//parameters:  none
//purpose: destroys "this" kerberosclass object
//---------------------------------------------------------------------------
//
```

```
kerberosclass::~kerberosclass(){};

//member functions
//---------------------------------------------------------------------------
//function: register_host(int  host_id,char* host_key)
//return value: none
//parameters: int host_id, char* host_key
//purpose: it adds an entry to the host_key database of kerberos; therefore, it
//      registers a host
//remarks: as we mentioned in the header file description the programmer has to
//      take care of the registration process in the driver of the simulation.
//      The programmer plays the role of the secure key distribution mean
//      between the kerberos and the host. The vector object's functionality
//      is used to add the entry to the database
//---------------------------------------------------------------------------
//
void kerberosclass::register_host(int  host_id,
                char* host_key)
{  //use vector's functionality
  host_key_database.add_host(host_id,host_key);
  return;
}//end register_host

//---------------------------------------------------------------------------
//function: remove_host(int host_id)
//return value: none
//parameters: int host_id
//purpose: it removes an entry from the host_key database of kerberos
//---------------------------------------------------------------------------
//
void kerberosclass::remove_host(int host_id)
{  //use vector's functionality
  host_key_database.remove_host(host_id);
  return;
}//end remove_host
```

```
//---------------------------------------------------------------------------
//function: register_user (char* user_id)
//return value: none
//parameters:  char* user_id
//purpose: it adds an entry (actually a char*) to the user database of kerberos;
//      therefore, it registers a user
//remarks: as we mentioned in the header file description, the programmer has to
//      take care of the user registration process in the driver of the
//      simulation.
//      The user registers in a "naive" way with kerberos using the user id.
//      We decided this only for simplification purposes for the
//      simulation. The vector object's member function "push_back(class T)"
//      is used to add the entry to the database
//---------------------------------------------------------------------------
//
void kerberosclass::register_user (char* user_id)
{ //use the vector functionality
  user_database.push_back(user_id);
  return ;
}//end register_user

//---------------------------------------------------------------------------
//function: remove_user (char* user_id)
//return value: none
//parameters:  char* user_id
//purpose: it removes an entry (actually a char*) from the user database of
//      kerberos. An iterator is used to traverse the database and the member
//      function of vector "erase(iterator i) is used to remove the vector
//      entry pointed by i.
//---------------------------------------------------------------------------
//
void kerberosclass::remove_user (char* user_id)
{
  //traverse the vector
  for (user_vector::iterator i= user_database.begin();
    i!=user_database.end(); i++){
    //if you find the user entry
    if ((*i)== user_id){
      user_database.erase(i);
    }//endif
  }//endfor
  return;
}//end remove_user

//---------------------------------------------------------------------------
//function: request_permission(char* user_id,int host_id)
//return value: bool
//parameters:  char* user_id,int host_id
//purpose: simulates the kerberos authentication process.
//      A user_id, at a host_id, requests permission to access the nacclass
//      object for further packet forwarding. If the host is registered and
//      the user is registered then the return value is "true" (the host has
//      permission).Otherwise "false".
//---------------------------------------------------------------------------
//
bool kerberosclass::request_permission(char* user_id,
                               int   host_id)
{
  bool user_exists = false;
  bool permission_granted = false;

  //first lookup the database of hosts
  //use the object's member function
  bool host_exists = host_key_database.is_host_registered(host_id);

  //then lookup the database of users
  //traverse the user database (vector) with an iterator
  for (user_vector::iterator i= user_database.begin();
    i!=user_database.end(); i++){
    //if you find the user registered
    if (strcmp((*i), user_id)==0){
      user_exists = true;
    }//end if
  }//end for

  //only if both are registered
  if (host_exists && user_exists){
    permission_granted = true;
  }//endif
  return permission_granted;
}//end request_permission

//---------------------------------------------------------------------------
//function: create_session_key()
//return value: char*
//parameters:  none
//purpose: to create the  session key the host needs to communicate with the nac.
//      In a more elaborate prototype a key_generator should be used.
//      Here we stub the key to the string:"sesskey".
//---------------------------------------------------------------------------
//
char* kerberosclass::create_session_key()
{
  return ("sesskey");
}//end create_session_key()

//---------------------------------------------------------------------------
//function: create_ticket(int  host_id, char* session_key)
//return value: a ticketclass object
//parameters:  int  host_id,char* session_key
//purpose: the host needs a ticket to be accepted by the nac.
//      Normally a ticket would have more information like a timestamp.
//      We simplify the ticket to be only the host_id and
//      a session_key. A ticketclass object is created and returned.
//
```

```
//--------------------------------------------------------------------------
//
ticketclass kerberosclass::create_ticket( int  host_id,
                                          char* session_key)
{
  //create a ticketClass object
  ticketclass current_ticket (host_id, session_key);

  return (current_ticket);
}//end create_ticket(int  host_id, char* session_key)

//--------------------------------------------------------------------------
//function: send_key_ticket(int host_id)
//return value: const char* (a file name)
//parameters:  int  host_id
//purpose: the kerberos creates a session key and a ticket for the host.
//      It then puts the ticket to file (TICKET_FILE).(This file should be
//      encrypted with the selected encryption metbod using the
//      KEY OF THE NAC that Kerberos knows, we don't do it here).
//      The name of the file and the session key are used to create
//      a key_ticket_class object that is put to a file (KEY_TICKET_FILE.
//      (This file should be encrypted with the selected encryption metbod
//      but using the KEY OF THE HOST that Kerberos knows, again we skip
//      this step). The name of the file is returned (to the host).
//--------------------------------------------------------------------------
//
const char* kerberosclass::send_key_ticket(int host_id)
{
  //create a session key
  char* session_key = create_session_key();

  //and a ticket to be send to nac
  ticketclass ticket = create_ticket(host_id ,session_key);

  //open a file to put the ticket
  fstream t_out(TICKET_FILE, ios::out);

  //now put the created ticket in this file
  //use the overloaded operator
  t_out << ticket;

  //for safety
  t_out.close();

  //apply encryption to the file, use NAC's key (NOT IMPLEMENTED):
  //encrypt(TICKET_FILE,"nackey")

  //create the key_ticket_class to be returned
  //contains the session_key and the ticket_file name
  key_ticket_class key_ticket(session_key,const_cast<char*>(TICKET_FILE));

  //open a file to put the key_ticket_class object
  fstream kt_out(KEY_TICKET_FILE, ios::out);
  //use the overloaded operator
  kt_out<<key_ticket;

  //apply encryption to the file use HOST's key (NOT IMPLEMENTED):
  //encrypt (TICKET_FILE,"hostkey")

  //for safety
  kt_out.close();

  //return the name of the file that contains all
  return (KEY_TICKET_FILE);
}//end send_key_ticket(int host_id)

//-----------------------------------------------------------------------------
//FUNCTIONS USED FOR DEBUGGING PURPOSES
//-----------------------------------------------------------------------------
//
```

```
//------------------------------------------------------------------------------
//function: print_users()
//------------------------------------------------------------------------------
//
void kerberosclass::print_users ()
{
  //traverse the vector
  for (user_vector::iterator i = user_database.begin();
    i!=user_database.end(); i++){
    //if you find the user entry
    cout<<(*i)<<endl;
  }//endfor
  return;
}//end print_users

//------------------------------------------------------------------------------
//function: print_hosts_keys()
//------------------------------------------------------------------------------
//
void kerberosclass::print_hosts_keys()
{
  cout<<host_key_database;
  return;
}//end print_hosts_keys


//end file kerberosclass.cpp
```

```
#ifndef __key_ticket_class_H__
#define __key_ticket_class_H__

//******************************************************************************
//******************************************************************************
// File : key_ticket_class.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:This is the class that contains the session key and the name of
//          the file that contains the ticket. The objects are created by
//          the kerberos class and are sent to the host that
//          requested for authentication and a session key to send traffic
//          to the nac. The necessary functions to access the data are also
//          provided.
//
// Assumptions: The file that contains the ticket is encrypted with the
//          encryption method chosen for the protocol (IDEA in our case)
//
//
//
//*******************************************************************************
//
#include <iostream>

class key_ticket_class {
  //overloading of operators
  friend ostream& operator<<(ostream&, const key_ticket_class&);
  friend istream& operator>>(istream&, key_ticket_class& );

public:
  //default constructor
  key_ticket_class();
  //another constructor
  explicit key_ticket_class(char* session_key,
                    char* ticket_file);
  //destructor
  ~key_ticket_class();

  //member functions
  char* session_key_is()const{return (session_key);}
  char* ticket_file_is()const{return (ticket_file);}

private:
  //data
  char* session_key;
  char* ticket_file;
};//end key_ticket_class

#endif
```

```
//************************************************************************
//************************************************************************
// File : key_ticket_class.cpp
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:same as in key_ticket_class.h file
// Assumptions:same as in key_ticket_class.h file
//
//
//
//************************************************************************
//
#include "key_ticket_class.h"

//----------------------------------------------------------------------------
//function: default constructor
//return value: a key_ticket_class object
//parameters: none
//purpose: the initial values are entered in order to be able to check the
//    existance of a key and a file name.The key_ticket_class object is
//    initialized with "notexists" strings
//----------------------------------------------------------------------------
//
key_ticket_class::key_ticket_class ():
session_key("notexists"),ticket_file("notexists"){}




//----------------------------------------------------------------------------
//function: another constructor
//return value: a key_ticket_class object
```

```
//parameters: char* session_key, char* ticket_file
//purpose: the session_key and the ticket_file name are entered to the created
//   object by kerberos. The object is going to be sent to the host for further
//   usage.
//---------------------------------------------------------------------------
//
key_ticket_class::key_ticket_class (char* session_key, char* ticket_file):
session_key(session_key),ticket_file(ticket_file){}

//---------------------------------------------------------------------------
//function: destructor
//return value: none
//parameters:  none
//purpose: destroys "this" key_ticket_class object
//---------------------------------------------------------------------------
//
key_ticket_class::~key_ticket_class(){}

//---------------------------------------------------------------------------
//function: ostream& operator<<(ostream& out, const key_ticket_class& MY_KT)
//return value:ostream&
//parameters:  ostream  out,key_ticket_class MY_KT
//purpose: a user defined class has to define how << behaves.
//        In our case if the key_ticket_class object passed in by ref
//        has not been initialized (session_key and ticket_file strings are
//        not valid) then we only inform the user that the object does not
//        exist, otherwise we display the two strings.
//
//remark: even one unitialized value would lead to an invalid object
//---------------------------------------------------------------------------
//
ostream& operator<<(ostream& out, const key_ticket_class& MY_KT)
{

  if ((MY_KT.session_key == "notexists")&&
        (MY_KT.ticket_file == "notexists")){
    //for some reason the key_ticket object is not there
    out << "<key_ticket object does not exist>"<<endl;
  }
  else if (MY_KT.session_key == "notexists"){
    //for some reason the key_ticket object's session_key is not there
    out<<"<key_ticket object.session_key does not exist> "<< endl;
  }
  else if (MY_KT.ticket_file=="notexists"){
    //for some reason the key_ticket object's ticket_file is not there
    out<<"<key_ticket object.ticket_file does not exist> "<< endl;
  }
  else{//everything exists
        out << MY_KT.session_key<<endl;
        out << MY_KT.ticket_file<<endl;
  }//end if
  return out;
}//end operator<<(ostream& out, const key_ticket_class& MY_KT)

//----------------------------------------------------------------------------
//function: istream& operator>>(istream& in, key_ticket_class& my_kt )
//return value:istream&
//parameters:  istream& in,key_ticket_class my_kt
//purpose: a user defined class has to define how >> behaves.
//      Here we input the two values to the key_ticket_class object
//       on the RHS
//----------------------------------------------------------------------------
//
istream& operator>>(istream& in, key_ticket_class& my_kt )
{
  in >>my_kt.session_key;
  in >>my_kt.ticket_file;
  return in;
}//end  operator>>(istream& in, key_ticket_class& my_kt )
```

```
//end file key_ticket_class.cpp
```

```
#ifndef __md5_H__
#define __md5_H__
/* MD5.H - header file for MD5C.C
 */

/* Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.

License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.

License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.
 */
#include "global.h"
/* MD5 context. */
typedef struct {
  UINT4 state[4];                          /* state (ABCD) */
  UINT4 count[2];      /* number of bits, modulo 2^64 (lsb first) */
  unsigned char buffer[64];                      /* input buffer */
} MD5_CTX;

void MD5Init /*PROTO_LIST*/ (MD5_CTX *);
void MD5Update /*PROTO_LIST*/
  (MD5_CTX *, unsigned char *, unsigned int);
void MD5Final /*PROTO_LIST*/ (unsigned char [16], MD5_CTX *);

#endif
```

```
/* MD5C.C - RSA Data Security, Inc., MD5 message-digest algorithm
 */

/* Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.

License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.

License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.
 */

#include "global.h"
#include "md5.h"

#include <memory.h>

/* Constants for MD5Transform routine.
 */
#define S11 7
#define S12 12
#define S13 17
```

```
#define S14 22
#define S21 5
#define S22 9
#define S23 14
#define S24 20
#define S31 4
#define S32 11
#define S33 16
#define S34 23
#define S41 6
#define S42 10
#define S43 15
#define S44 21

static void MD5Transform /*PROTO_LIST*/ (UINT4 [4], unsigned char [64]);
static void Encode /*PROTO_LIST*/
 (unsigned char *, UINT4 *, unsigned int);
static void Decode /*PROTO_LIST*/
 (UINT4 *, unsigned char *, unsigned int);
static void MD5_memcpy /*PROTO_LIST*/ (POINTER, POINTER, unsigned int);
static void MD5_memset /*PROTO_LIST*/ (POINTER, int, unsigned int);

static unsigned char PADDING[64] = {
 0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};

/* F, G, H and I are basic MD5 functions.
 */
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))
```

```
/* ROTATE_LEFT rotates x left n bits.
 */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))

/* FF, GG, HH, and II transformations for rounds 1, 2, 3, and 4.
Rotation is separate from addition to prevent recomputation.
 */
#define FF(a, b, c, d, x, s, ac) { \
 (a) += F ((b), (c), (d)) + (x) + (UINT4)(ac); \
 (a) = ROTATE_LEFT ((a), (s)); \
 (a) += (b); \
  }
#define GG(a, b, c, d, x, s, ac) { \
 (a) += G ((b), (c), (d)) + (x) + (UINT4)(ac); \
 (a) = ROTATE_LEFT ((a), (s)); \
 (a) += (b); \
  }
#define HH(a, b, c, d, x, s, ac) { \
 (a) += H ((b), (c), (d)) + (x) + (UINT4)(ac); \
 (a) = ROTATE_LEFT ((a), (s)); \
 (a) += (b); \
  }
#define II(a, b, c, d, x, s, ac) { \
 (a) += I ((b), (c), (d)) + (x) + (UINT4)(ac); \
 (a) = ROTATE_LEFT ((a), (s)); \
 (a) += (b); \
  }

/* MD5 initialization. Begins an MD5 operation, writing a new context.
 */
void MD5Init (context)
MD5_CTX *context;                          /* context */
{
  context->count[0] = context->count[1] = 0;
  /* Load magic initialization constants.
```

```
*/
 context->state[0] = 0x67452301;
 context->state[1] = 0xefcdab89;
 context->state[2] = 0x98badcfe;
 context->state[3] = 0x10325476;
}

/* MD5 block update operation. Continues an MD5 message-digest
 operation, processing another message block, and updating the
 context.
 */
void MD5Update (context, input, inputLen)
MD5_CTX *context;                          /* context */
unsigned char *input;                   /* input block */
unsigned int inputLen;             /* length of input block */
{
 unsigned int i, index, partLen;

 /* Compute number of bytes mod 64 */
 index = (unsigned int)((context->count[0] >> 3) & 0x3F);

 /* Update number of bits */
 if ((context->count[0] += ((UINT4)inputLen << 3))
  < ((UINT4)inputLen << 3))
context->count[1]++;
 context->count[1] += ((UINT4)inputLen >> 29);

 partLen = 64 - index;

 /* Transform as many times as possible.
*/
 if (inputLen >= partLen) {
MD5_memcpy
 ((POINTER)&context->buffer[index], (POINTER)input, partLen);
MD5Transform (context->state, context->buffer);
```

```
 for (i = partLen; i + 63 < inputLen; i += 64)
   MD5Transform (context->state, &input[i]);

 index = 0;
  }
  else
 i = 0;

  /* Buffer remaining input */
  MD5_memcpy
 ((POINTER)&context->buffer[index], (POINTER)&input[i],
  inputLen-i);
}

/* MD5 finalization. Ends an MD5 message-digest operation, writing the
  the message digest and zeroizing the context.
 */
void MD5Final (digest, context)
unsigned char digest[16];                    /* message digest */
MD5_CTX *context;                                /* context */
{
  unsigned char bits[8];
  unsigned int index, padLen;

  /* Save number of bits */
  Encode (bits, context->count, 8);

  /* Pad out to 56 mod 64.
*/
  index = (unsigned int)((context->count[0] >> 3) & 0x3f);
  padLen = (index < 56) ? (56 - index) : (120 - index);
  MD5Update (context, PADDING, padLen);

  /* Append length (before padding) */
 MD5Update (context, bits, 8);
 /* Store state in digest */
 Encode (digest, context->state, 16);

 /* Zeroize sensitive information.
*/
 MD5_memset ((POINTER)context, 0, sizeof (*context));
}

/* MD5 basic transformation. Transforms state based on block.
 */
static void MD5Transform (state, block)
UINT4 state[4];
unsigned char block[64];
{
 UINT4 a = state[0], b = state[1], c = state[2], d = state[3], x[16];

 Decode (x, block, 64);

 /* Round 1 */
 FF (a, b, c, d, x[ 0], S11, 0xd76aa478); /* 1 */
 FF (d, a, b, c, x[ 1], S12, 0xe8c7b756); /* 2 */
 FF (c, d, a, b, x[ 2], S13, 0x242070db); /* 3 */
 FF (b, c, d, a, x[ 3], S14, 0xc1bdceee); /* 4 */
 FF (a, b, c, d, x[ 4], S11, 0xf57c0faf); /* 5 */
 FF (d, a, b, c, x[ 5], S12, 0x4787c62a); /* 6 */
 FF (c, d, a, b, x[ 6], S13, 0xa8304613); /* 7 */
 FF (b, c, d, a, x[ 7], S14, 0xfd469501); /* 8 */
 FF (a, b, c, d, x[ 8], S11, 0x698098d8); /* 9 */
 FF (d, a, b, c, x[ 9], S12, 0x8b44f7af); /* 10 */
 FF (c, d, a, b, x[10], S13, 0xffff5bb1); /* 11 */
 FF (b, c, d, a, x[11], S14, 0x895cd7be); /* 12 */
 FF (a, b, c, d, x[12], S11, 0x6b901122); /* 13 */
 FF (d, a, b, c, x[13], S12, 0xfd987193); /* 14 */
 FF (c, d, a, b, x[14], S13, 0xa679438e); /* 15 */
FF (b, c, d, a, x[15], S14, 0x49b40821); /* 16 */

/* Round 2 */
GG (a, b, c, d, x[ 1], S21, 0xf61e2562); /* 17 */
GG (d, a, b, c, x[ 6], S22, 0xc040b340); /* 18 */
GG (c, d, a, b, x[11], S23, 0x265e5a51); /* 19 */
GG (b, c, d, a, x[ 0], S24, 0xe9b6c7aa); /* 20 */
GG (a, b, c, d, x[ 5], S21, 0xd62f105d); /* 21 */
GG (d, a, b, c, x[10], S22,  0x2441453); /* 22 */
GG (c, d, a, b, x[15], S23, 0xd8a1e681); /* 23 */
GG (b, c, d, a, x[ 4], S24, 0xe7d3fbc8); /* 24 */
GG (a, b, c, d, x[ 9], S21, 0x21e1cde6); /* 25 */
GG (d, a, b, c, x[14], S22, 0xc33707d6); /* 26 */
GG (c, d, a, b, x[ 3], S23, 0xf4d50d87); /* 27 */
GG (b, c, d, a, x[ 8], S24, 0x455a14ed); /* 28 */
GG (a, b, c, d, x[13], S21, 0xa9e3e905); /* 29 */
GG (d, a, b, c, x[ 2], S22, 0xfcefa3f8); /* 30 */
GG (c, d, a, b, x[ 7], S23, 0x676f02d9); /* 31 */
GG (b, c, d, a, x[12], S24, 0x8d2a4c8a); /* 32 */

/* Round 3 */
HH (a, b, c, d, x[ 5], S31, 0xfffa3942); /* 33 */
HH (d, a, b, c, x[ 8], S32, 0x8771f681); /* 34 */
HH (c, d, a, b, x[11], S33, 0x6d9d6122); /* 35 */
HH (b, c, d, a, x[14], S34, 0xfde5380c); /* 36 */
HH (a, b, c, d, x[ 1], S31, 0xa4beea44); /* 37 */
HH (d, a, b, c, x[ 4], S32, 0x4bdecfa9); /* 38 */
HH (c, d, a, b, x[ 7], S33, 0xf6bb4b60); /* 39 */
HH (b, c, d, a, x[10], S34, 0xbebfbc70); /* 40 */
HH (a, b, c, d, x[13], S31, 0x289b7ec6); /* 41 */
HH (d, a, b, c, x[ 0], S32, 0xeaa127fa); /* 42 */
HH (c, d, a, b, x[ 3], S33, 0xd4ef3085); /* 43 */
HH (b, c, d, a, x[ 6], S34,  0x4881d05); /* 44 */
HH (a, b, c, d, x[ 9], S31, 0xd9d4d039); /* 45 */
HH (d, a, b, c, x[12], S32, 0xe6db99e5); /* 46 */
```

```
  HH (c, d, a, b, x[15], S33, 0x1fa27cf8); /* 47 */
  HH (b, c, d, a, x[ 2], S34, 0xc4ac5665); /* 48 */

  /* Round 4 */
  II (a, b, c, d, x[ 0], S41, 0xf4292244); /* 49 */
  II (d, a, b, c, x[ 7], S42, 0x432aff97); /* 50 */
  II (c, d, a, b, x[14], S43, 0xab9423a7); /* 51 */
  II (b, c, d, a, x[ 5], S44, 0xfc93a039); /* 52 */
  II (a, b, c, d, x[12], S41, 0x655b59c3); /* 53 */
  II (d, a, b, c, x[ 3], S42, 0x8f0ccc92); /* 54 */
  II (c, d, a, b, x[10], S43, 0xffeff47d); /* 55 */
  II (b, c, d, a, x[ 1], S44, 0x85845dd1); /* 56 */
  II (a, b, c, d, x[ 8], S41, 0x6fa87e4f); /* 57 */
  II (d, a, b, c, x[15], S42, 0xfe2ce6e0); /* 58 */
  II (c, d, a, b, x[ 6], S43, 0xa3014314); /* 59 */
  II (b, c, d, a, x[13], S44, 0x4e0811a1); /* 60 */
  II (a, b, c, d, x[ 4], S41, 0xf7537e82); /* 61 */
  II (d, a, b, c, x[11], S42, 0xbd3af235); /* 62 */
  II (c, d, a, b, x[ 2], S43, 0x2ad7d2bb); /* 63 */
  II (b, c, d, a, x[ 9], S44, 0xeb86d391); /* 64 */

  state[0] += a;
  state[1] += b;
  state[2] += c;
  state[3] += d;

  /* Zeroize sensitive information.
 */
  MD5_memset ((POINTER)x, 0, sizeof (x));
}

/* Encodes input (UINT4) into output (unsigned char). Assumes len is
  a multiple of 4.
 */
static void Encode (output, input, len)
```

```
unsigned char *output;
UINT4 *input;
unsigned int len;
{
  unsigned int i, j;

  for (i = 0, j = 0; j < len; i++, j += 4) {
 output[j] = (unsigned char)(input[i] & 0xff);
 output[j+1] = (unsigned char)((input[i] >> 8) & 0xff);
 output[j+2] = (unsigned char)((input[i] >> 16) & 0xff);
 output[j+3] = (unsigned char)((input[i] >> 24) & 0xff);
  }
}

/* Decodes input (unsigned char) into output (UINT4). Assumes len is
  a multiple of 4.
 */
static void Decode (output, input, len)
UINT4 *output;
unsigned char *input;
unsigned int len;
{
  unsigned int i, j;

  for (i = 0, j = 0; j < len; i++, j += 4)
 output[i] = ((UINT4)input[j]) | (((UINT4)input[j+1]) << 8) |
   (((UINT4)input[j+2]) << 16) | (((UINT4)input[j+3]) << 24);
}

/* Note: Replace "for loop" with standard memcpy if possible.
 */

static void MD5_memcpy (output, input, len)
POINTER output;
POINTER input;
```

```
unsigned int len;
{
/*
 unsigned int i;

 for (i = 0; i < len; i++)
 output[i] = input[i];
*/
 memcpy(output,input,len);
}

/* Note: Replace "for loop" with standard memset if possible.
 */
static void MD5_memset (output, value, len)
POINTER output;
int value;
unsigned int len;
{
/*
 unsigned int i;

 for (i = 0; i < len; i++)
 ((char *)output)[i] = (char)value;
*/
 memset(output,value,len);
}
```

```
#ifndef __nacclass_H__
#define __nacclass_H__

//*************************************************************************
//*************************************************************************
// File : nacclass.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description: the nacclass simulates the functionality of our firewall-like
//          network access controller and demos the protocol that we propose
//          Nac has the functionality to check if a host is authorized by
//          kerberos to send packets by examining the host's ticket. It then
//          creates a tableclass object (tag_key table), puts a copy in a
//          file encrypts it with the session_key that was retrieved from the
//          ticket and returns the filename of the (encrypted) file to the
//          host.Nacclass object puts the host_id and the tableclass object
//          in the hosts_tables_db database for further usage.
//          The nac authenticates the ip_packets that arrive.First a mac is
//          calculated for the message of the ip_packet. The mac is
//          calculated after the (ip_packet's) ki corresponding key is
//          appended to the message. By comparing the two macs (calculated
//          and the one contained in the ip_packet) the nac decides if the
//          ip_packet is authorized for forwarding.
//
// Assumptions: none
//
//
//
//*************************************************************************
//
#include "ip_packetclass.h"
#include "tableclass.h"
```

```
#include "ticketclass.h"
#include "host_table_vector.h"
#include "constants.h"
#include "host_table_class.h"
#include "host_key_class.h"
#include "host_ticket_class.h"
#include "fileutils.h"
#include <iostream>
#include <fstream>


extern const char* TABLE_FILE;
extern const char* HOST_TICKET_FILE;

class nacclass{

public:
  //constractors destructors
  //default constractor
  nacclass();

   //destructor
  ~nacclass();

  //member functions
  //-------------------------------------------------------------------------
  //a host asks for a table by sending the filename of the(encrypted) file that
  //contains the ticket
  //postcondition: the filename of the (encrypted) file that contains the
  //    tableclass object is returned
  //-------------------------------------------------------------------------
  char* prepare_table(const char*);
  //-------------------------------------------------------------------------
  //a packet arrives and needs to be authenticated
  //postcondition: if the mac of the packet is the same with the mac that nac
  //        calculates for the packet the packet is authenticated and
  //        "true" is returned. Otherwise "false"
  //------------------------------------------------------------------------
  bool authenticate_packet(ip_packetclass ip_packet);

private:
  //data
  //the database vector of hosts and tables
  host_table_vector hosts_tables_db;

          //auxilliary functions
  //for the  authentication
  //NOT NEEDED TAKE OUT AT CLEANUP
  char* generate_mac();
  bool compare_macs(char* in_mac ,char* calculated_mac);

  //add to database of hosts and tables
  void add_host_table (int host, tableclass* tableptr);

};//end nac class

#endif

//end file nacclass.h
```

```
//****************************************************************************
//****************************************************************************
// File : nacclass.cpp
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description: the same as in nacclass.h file
//
// Assumptions
//
// Warnings
//
//****************************************************************************
//
#include "nacclass.h"
extern "C"{
#include "hashutils.h"
#include "md5.h"
}
#include <stdio.h>
//---------------------------------------------------------------------------
//function: default constructor
//return value: a nacclass object
//parameters: none
//purpose: an empty database of hosts_tables is created
//---------------------------------------------------------------------------
//
nacclass::nacclass(){}

//---------------------------------------------------------------------------
//function: destructor
//return value: none
//parameters:  none
```

```
//purpose: destroys "this" nacclass object
//-----------------------------------------------------------------------------
//
nacclass::~nacclass(){}

//member functions
//-----------------------------------------------------------------------------
//function: prepare_table(const char* HOST_TICKET_FILE)
//return value: char*
//parameters:  const char* HOST_TICKET_FILE
//purpose: a host asks for a table by sending the file name of the file that
//        contains the host_ticket_class object
//        the nacclass object (nac) reads the host_id and the ticketfile name
//        from the HOST_TICKET_FILE file. NAC decrypts the ticketfile and gets
//        the host_id and the session_key(as they were encrypted by kerberos).
//        If the host_id send by the host is the same with the host_id that was
//        retrieved from the ticket: NAC prepares a new tableclass object, adds
//        an entry for the host to the hosts_tables database, puts the table to
//        a file, encrypts it with the session_key (retrieved from the ticket)
//        and returns the filename of the encrypted file to the host. Otherwise
//        if the host is not verified by the ticket the string "invalid_ticket"
//        is returned.
//-----------------------------------------------------------------------------
//
char* nacclass::prepare_table(const char* HOST_TICKET_FILE)
{
  char* retname = "invalid_ticket";

  //FIRST need to read from the file the host_id and
  //the ticket_file name
  host_ticket_class host_ticket;
  ticketclass host_key;

  //open the file for input
  fstream in(HOST_TICKET_FILE, ios::in);
```

```
//get the host_ticket_class object
in >> host_ticket;

//get the data from the host_ticket_class object
int host_id = host_ticket.host_id_is();
char* ticket_file = host_ticket.ticket_file_is();
//no need this stream anymore
in.close();

//decrypt the ticket file using the nackey
//decrypt (ticket_file,"nackey");

//from the decrypted file
//open it for input
fstream t_in(ticket_file, ios::in);
//read the host id and the sessionkey in the host_key_class object
t_in >> host_key;

//from the host_key_class object
//get the data needed to verify the host
int ticket_host_id = host_key.host_id_is();

//use it if encryption is integrated
//char* session_key = host_key.session_key_is();

//only if host is verified by the ticket
if (host_id == ticket_host_id){
  //dynamically allocate space for a tableclass object
  tableclass* new_table_ptr= new tableclass();

  //initialize the key table
  new_table_ptr -> initialize_the_key_pool(host_id);

  //update the database with the new entry
  add_host_table(host_id,new_table_ptr);
```

```
    //prepare the table file to be sent to the host
    fstream out(TABLE_FILE, ios::out);
    out<<(*new_table_ptr);
    out.close();

    //encrypt the file using the "session key"
    //encrypt(TABLE_FILE, session_key);

    //this file name prepare to return
    retname = const_cast<char*>(TABLE_FILE);
    }//endif
  return retname;
}//end prepare_table

//--------------------------------------------------------------------------
//function: authenticate_packet(ip_packetclass ip_packet)
//return value: bool
//parameters:  ip_packetclass ip_packet
//purpose: nac extracts the filename that contains the (actual)message
//      The contents of this file are copied to the a stream buffer
//      The ki and the host_id are retrieved from the ip_packetclass object
//      using the ip_packetclass functionality. Nac checks to see if the host
//      that created the ip_packet is authorized to send packets (if an entry
//      exists for this host in the hosts_tables_db). If this is "true" (a
//      table exists for the host) the key for the relevant ki is retrieved
//      and the ki and the key is APPENDED to the stream buffer file.
//      MD5 is called for the FILE_TO_HASH and the 128 bit (16 byte) mac is
//      calculated. If the calculated mac is the same with the ip_packet mac
//      the ip_packet is authenticated to go and the function returns "true"
//      otherwise "false".
//
//--------------------------------------------------------------------------
//
bool nacclass::authenticate_packet(ip_packetclass ip_packet)
```

```
{
  bool authenticated = false;
  //the message data is in this file
  char* message_file = ip_packet.filename_is();
  //a pointer to a buffer that will carry the contents of the message
  char* temp_buffer = new char[MAX_MESSAGE_SIZE] ;
  char* message_buffer;

  int ki = ip_packet.ki_is();
  int host_id= ip_packet.host_is();
  int key = 0;

  if(hosts_tables_db.is_host_authorized(host_id)){
    key = hosts_tables_db.
    key_for_host_ki(host_id,ki);
  }//end if

  //the key has to be appended in the next line to the message
  temp_buffer = copytobuffer(message_file,temp_buffer,ki,key);
  message_buffer = new char [strlen(temp_buffer)+1];
  strcpy (message_buffer, temp_buffer);
  delete[] temp_buffer;

  //md5 HAS TO BE CALLED FOR THE WHOLE THING
  unsigned char* mac = MD5String(message_buffer);

  if (compare2Digest(mac,ip_packet.mac_is())){
    authenticated = true;
  }
  return authenticated;
}//end authenticate_packet(ip_packetclass ip_packet)

//---------------------------------------------------------------------------
//function: add_host_table (int host, tableclass * tableptr)
//return value: none
```

```
//parameters: int host, tableclass * tableptr
//purpose: it adds an entry to the hosts_tables_db database of nac.
//      First a host_table_class object is created then the function uses
//      the functionality of the host_table_vector to add the new entry to
//      the hosts_tables_db.
//---------------------------------------------------------------------------
//
void nacclass::add_host_table (int host, tableclass * tableptr)
{
  host_table_class new_host_table(host,tableptr);
  hosts_tables_db.add_host_table(new_host_table);
  return ;
}//

//end file nacclass.cpp
```

```
#ifndef __tableclass_H__
#define __tableclass_H__
//****************************************************************************
//****************************************************************************
// File : tableclass.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:The table of (key_index,key) pairs created by nac and sent
//          to the host. The STL::map data structure was chosen to contain
//          the pairs of ki and key. The map"key" (first of the pair) is
//          the ki and the value (second of the pair) is the key. We chose
//          the key to be an integer value created by the srand function of
//          C++ with seed the host_id (for uniqueness of the random values).
//          This is a "naive" approach and was chosen for simplicity for demo
//          purposes. In a more serious implementation a key_generator must
//          be used and the key maybe chosen to be other than an integer
//          (a string maybe).
//            The class contains a map of NUM_PAIRS = 10 ki-key pairs and
//          the necessary functions to get the data. The auxilliary
//          functions are private so only the nac can get the data.
//            TO DO: CONSIDER USING RANDOM kis ALSO,DOES NOT AFFECT THE MAP,
//                 NOT NECESSARY
//
// Assumptions: we assume only authorized users will use the << and >> operator
//          that we defined
//
// Warnings
//
//****************************************************************************
#include <stdlib>
#include <map>
#include <iostream>
```

```
#include "constants.h"

using namespace std;

//a map is used for the tag_key pairs
typedef map<int ,int ,less<int> > table_of_ki_key_pairs;



class tableclass{

  //friend
  //we want nac to access private part
  friend class nacclass;
  //and the database of the nac
  friend class host_table_vector;

  //i/o stream operators overloading
  friend ostream& operator<<(ostream &, tableclass& );
  friend istream& operator>>(istream &, tableclass& );

public:
          //constractors destructors
  //default constractor
  tableclass();

  //copy constructor
  tableclass (const tableclass &);

  //destructor
  ~tableclass();

  //member functions
  //a host asks for the next ki_key pair from the table
  int get_next_ki ();
  int get_next_key(int ki)const;

  //not really needed, I put it for Debuging
   void print_table();

private:
  //DATA
  //a std::map
  table_of_ki_key_pairs ki_key_table;

  //initialize the tag key pool
  //use as seed the host_id for uniqueness
  void tableclass::initialize_the_key_pool(int host_id);

  //service intented for the NAC to look_up the table to find a key
  int look_up_table_for_key(int ki)const;

};//end table class
#endif
//end file tableclass.h
```

```
//******************************************************************************
//******************************************************************************
// File : tableclass.cpp
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description: same as in the header file
//
// Assumptions:we assume only authorized users will use the << and >> operator
//          that we defined
//
// Warnings
//
//******************************************************************************
//
#include "tableclass.h"

//----------------------------------------------------------------------------
//function: default constructor
//return value: a tableclass object
//parameters: none
//purpose: the initial values are entered in order to be able to check the
//existance of a key.the table is initialized with 0's
//----------------------------------------------------------------------------
//
tableclass::tableclass()
{
  //just create a zeroized object
  for (int i = 1; i<= NUM_PAIRS; i++){
    ki_key_table[i]=0;
  }
} //end constructor
```

```
//---------------------------------------------------------------------------
//function: copy constructor
//return value: a tableclass object
//parameters:  NEW_TABLE, a const tableclass object passed by reference;
//purpose: the creation of a table object with the same values as
//       the NEW_TABLE object
//WARNING: if you change the ki allocation from sequential to random
//       with the use of a ki_generator this WOULD NOT WORK instead
//       you should use iterators to copy the NEW_TABLE object to the
//       newly created table
//---------------------------------------------------------------------------
//
tableclass::tableclass (const tableclass &NEW_TABLE)
{
  for (int i = 1; i<= NUM_PAIRS; i++){
    ki_key_table[i] = NEW_TABLE.ki_key_table[i];
  }
}//end copy constructor

//---------------------------------------------------------------------------
//function: destructor
//return value: none
//parameters:  none
//purpose: destroys "this" table class object
//---------------------------------------------------------------------------
//
tableclass::~tableclass(){}

//member functions
//---------------------------------------------------------------------------
//function: initialize_the_key_pool(int host_id)
//return value: none
//parameters: int host_id, to be used as seed
//purpose:  This function is a service to the nac.
//       The key values are created using the srand function in C++ and then
//      they are entered to the map. The "key" to the map is the ki.
//remarks: as we mentioned in the header file description a key generator
//      should be chosen for a more elaborate approach. Also here the kis
//      are sequential from 1 to NUM_PAIR.It may look more elegant to
//      use a ki_generator for that purpose also. These more elaborate
//      solutions would not affect the functionality of our object. We chose
//      the simple approach for this demo.
//-----------------------------------------------------------------------------
//
void tableclass::initialize_the_key_pool(int host_id)
{
  //initializing the map
  //use the host to seed rand for recognition purposes
  srand(static_cast<unsigned int>(host_id));

  for (int i = 1; i<= NUM_PAIRS; i++){
    ki_key_table[i]= rand();
  }

  return;
}//end initialize_the_key_pool(int host_id)

//-----------------------------------------------------------------------------
//function: get_next_ki
//return value: int,the next ki
//parameters:  none
//purpose: service provided to a host,
//      the host asks for the next ki from the table, the next ki in the map
//      is chosen.
//WARNING:
//-----------------------------------------------------------------------------
//
int tableclass::get_next_ki ()
{
  //pick the next ki for the next series of packets
  static int current_ki = 0;

  current_ki++;//start from 1

  //the mod forces current_ki to be in the range or 0-(NUM_PAIRS-1)
  //so +1 corrects it to 1 - NUM_PAIRS
  if (current_ki%(NUM_PAIRS+1) == 0){//the whole range was covered
    //reset the counter
    current_ki=1;
  }

  return current_ki;
}//end get_next_ki()

//---------------------------------------------------------------------------
//function: get_next_key(int ki)
//return value: int,the next key
//parameters:  none
//purpose: service provided to a host,
//        the host asks for the next key from the table for a given ki
//        the logic simply follows from the above
//---------------------------------------------------------------------------
//
int tableclass::get_next_key(int ki)const
{
 return ki_key_table[ki];
}//end get_next_key

//---------------------------------------------------------------------------
//function: look_up_table_for_key(int ki)
//return value: int, the key corresponding to this ki
//parameters:  int ki
//purpose: services provided to the NAC
//        friend class nacclass to the  table object requests to find
//        the key for the passed in ki entry
```

```
//----------------------------------------------------------------------------
//
int tableclass::look_up_table_for_key(int ki)const
{
  return ki_key_table[ki];
}//end look_up_table_for_key(int ki)

//----------------------------------------------------------------------------
//function: print_table()
//return value: none
//parameters:  none
//purpose: FOR DEBUG PURPOSES (NOT ACTUALLY NEEDED)
//----------------------------------------------------------------------------
//
void tableclass::print_table()
{
  if (ki_key_table.size() > 0){
    cout<< "ki    key" <<endl;
    for (table_of_ki_key_pairs::iterator i = ki_key_table.begin();
      i != ki_key_table.end();i++){
        cout<< (*i).first<<"     "<< (*i).second<<endl;
      }//end for
  }
  else cout <<" The table does not exist."<<endl;
  return;
}//end print_table

//----------------------------------------------------------------------------
//function: ostream& operator<<(ostream & out, tableclass& MY_TABLE)
//return value:ostream&
//parameters:  ostream  out,tableclass MY_TABLE
//purpose: a user defined class has to define how << behaves.
//       In our case if the table size passed in by ref has 0 size
//       then we only inform the user that the
//       table does not exist, otherwise we display the two values
//      of each pair in the map using an iterator.
//warning: DOES NOT HAVE to change when the kis are not in the range
//      1-NUM_PAIRS but are selected randomly
//       ALSO no need to change anything if we change the key type,as
//      long as << works for that type (native; we must take care of it
//      if we use a user defined type)
//remark: MY_TABLE here is not const as it should because the compiler does not
//     allow the iterators to be declared
//---------------------------------------------------------------------------
//
ostream& operator<<(ostream & out, tableclass& MY_TABLE)
{
  if (MY_TABLE.ki_key_table.size() <= 0){
    //for some reason the table is not there
    out<<"<table does not exist> "<< endl;
  }
  else{
    for(table_of_ki_key_pairs::iterator i =
      MY_TABLE.ki_key_table.begin();
      i != MY_TABLE.ki_key_table.end();i++){
        //line by line for easing sequential file readout
        out<< (*i).first<<endl;
        out<< (*i).second<<endl;
    }//end for
  }//endif
  return(out);
}//end <<() overloading

//---------------------------------------------------------------------------
//function: istream& operator>>(istream & in, tableclass& my_table)
//return value:istream&
//parameters:  istream in,tableclass my_table
//purpose: a user defined class has to define how >> behaves.
//      Here we input the table pairs to the tableclass object on the RHS
//warning: DOES NOT HAVE to change when the kis are not in the range
```

```
//        1-NUM_PAIRS but are selected randomly
//         ALSO no need to change anything if we change the key type,as
//        long as >> works for that type (native); we must take care of it
//        if we use a user defined type
//--------------------------------------------------------------------------------
//
istream& operator>>(istream & in, tableclass& my_table)
{
  for (table_of_ki_key_pairs::iterator i =
    my_table.ki_key_table.begin();
    i != my_table.ki_key_table.end();i++){
    in>> (*i).first;
    in>> (*i).second ;
  }
  return(in);
}//end >>() overloading
//end file tableclass.cpp
```

```
#ifndef __ticketclass_H__
#define __ticketclass_H__

//*************************************************************************
//*************************************************************************
// File : ticketclass.h
// Name : Ioannis Kondoulis
//
// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description:This is the ticket created by kerberos and send by the host
//          to the nac. The class contains the host_id and the session_key
//          as well as the necessary functions to get the data.The auxilliary
//          functions are private so only the nac can get the data. No other
//          class can access the ticket data.
//
// Assumptions: only authorized users can use the operators << and >>
//
//
//
//*************************************************************************
//
#include <iostream>

class ticketclass{
  //friend
  //we want nac to access private part
  friend class nacclass;
  //i/o stream operator overloading
  friend ostream& operator<<(ostream &,const ticketclass&);
  friend istream& operator>>(istream &,ticketclass&);

public:
  //default constructor
  ticketclass();

  //for a session with the nac the host must use the ticket
  // and the session key created by the kerberos
  //another constructor
  explicit ticketclass (int host_id , char* session_key);

  //copy const
  ticketclass (const ticketclass & );

  //destructor
  ~ticketclass();

private:
  //DATA
  int host_id;
  char* session_key;

  //AUXILIARY FUNCTIONS
  //made private to be used by friend nac class
  //to get the info needed
          int host_id_is()const;
  char* session_key_is()const;
};//end ticketclass

#endif
//end file ticketclass.h
```

```
//****************************************************************************
//****************************************************************************
// File : ticketclass.cpp
// Name : Ioannis Kondoulis
//

// Operating Enviroment: MS Windows NT 4.0
// compiler: Borland C++ for Windows, ver 5.0
// Date: 19 September 1998
// Description: same as in the header file
//
// Assumptions: we assume only authorized users will use the << and >> operator
//          that we defined
//
//
//****************************************************************************

#include "ticketclass.h"
#include <iostream>

//---------------------------------------------------------------------------
//function: default constructor
//return value: a ticket class object
//parameters: none
//purpose: the initial values are entered in order to be able to check the
//existance of a key
//---------------------------------------------------------------------------
//
ticketclass::ticketclass():host_id(-1),session_key("no_key"){}

//---------------------------------------------------------------------------
//function: another constructor
//return value: a ticket class object
//parameters: host_id, session_key
//purpose:a ticket object for host:host_id
```

```
//      the session_key value has been created by a kerberos object
//---------------------------------------------------------------------------
//
ticketclass::ticketclass (int host_id,char* session_key):
  host_id(host_id),session_key(session_key){}

//---------------------------------------------------------------------------
//function: copy constructor
//return value: a ticket class object
//parameters:  new_ticket, a ticket class object passed by reference;
//purpose: the creation of a ticket object with the same values as
//        the new_ticket object
//---------------------------------------------------------------------------
//
ticketclass::ticketclass(const ticketclass & new_ticket)
{
  host_id = new_ticket.host_id;
  session_key = new_ticket.session_key;
}

//---------------------------------------------------------------------------
//function: destructor
//return value: none
//parameters:  none
//purpose: destroys "this" ticket class object
//---------------------------------------------------------------------------
//
ticketclass::~ticketclass(){}


//AUXILLIARY functions

//---------------------------------------------------------------------------
//function: host_id_is
//return value: the host_id(int) of this object
```

```
//parameters:  none
//purpose: friend class (nac class) to the  ticket object can read for which
//        host this ticket is. The function can read the value but cannot
//        change it (const)
//---------------------------------------------------------------------------
//
int ticketclass::host_id_is()const
{
  return host_id;
}//end host_id_is

//---------------------------------------------------------------------------
//function: session_key_is
//return value: the session_key (int) of this object
//parameters:  none
//purpose: friend class (nac class) to the  ticket object can read the
//        session_key included in this ticket.The function can read the
//        value but cannot change it (const)
//---------------------------------------------------------------------------
//
char* ticketclass::session_key_is()const
{
  return session_key;
}//end session_key_is

//---------------------------------------------------------------------------
//function: ostream& operator<<(ostream & out, const ticketclass& MY_TICKET)
//return value:ostream&
//parameters:  ostream & out,ticketclass& MY_TICKET
//purpose: a user defined class has to define how << behaves.
//        In our case if the host_id=-1 then we only inform the user that the
//        ticket does not exist (because it is not properly initialized)
//        otherwise we display the two values.)
//---------------------------------------------------------------------------
//
```

```
ostream& operator<<(ostream & out, const ticketclass& MY_TICKET)
{
  if (MY_TICKET.host_id <0){
    //no ticket has been assigned
    out<<"<ticket does not exist> "<< endl;
  }
  else {
    out << MY_TICKET.host_id <<endl;
    out << MY_TICKET.session_key<<endl;
  }
  return(out);
}//end <<() overloading



//---------------------------------------------------------------------------
//function: istream& operator>>(istream & in, ticketclass& my_ticket)
//return value:istream&
//parameters:  istream & in,ticketclass& my_ticket
//purpose: a user defined class has to define how >> behaves.
//      Here we input the ticket values to ticketclass object on the RHS
//---------------------------------------------------------------------------
//
istream& operator>>(istream & in, ticketclass& my_ticket)
{
  in >> my_ticket.host_id ;
  in >> my_ticket.session_key;

  return(in);
}//end >>() overloading

//end file ticketclass.cpp
```

APPENDIX D. DOCUMENTATION FOR MD5 PERFORMANCE TEST CODE

1. Description

The program measures the throughput of the MD5 message digest algorithm. The program uses the services of the "buffercreate.h" file. Main() creates a series of eight messages with sizes 512 bytes, 1, 2, 4, 8, 16, 32 and 64 Kbytes and applies MD5 on them. To calculate an average value for the throughput we calculate the MD5 of each of these messages 1000 times.

2. Layers

The main uses three function to measure the throughput:

Create_buffer: receives the ostrstream object (by reference) and a float that is the size of the message that we want to be created and it floods the object with the char "a" up to the size that was passed in as the second argument.

Calculate_time: receives an string (char*) and calculates how long it takes (in msecs) to apply MD5 on it 1000 times. It returns the calculated time.

calculate_throughput: the time that took to apply MD5 1000 times on a string and the size of the string and calculates the throughput. The result is in Mbits/sec

3. Modules

3.1 main

3.1.1 DESCRIPTION: The main() declares eight char*s and eight ostrstream output objects. The objects are declared without arguments therefore they dynamically allocate as many memory is required for the data that are going to be output to them. They stop allocating memory only when we "freeze" them using the .str() member function. Then the messages (buffers) are filled by calling the create_buffer( ) function and are frozen. For each of them we calculate the time that it takes for executing 1000 the MD5String() function and the throughput.

3.1.2 DATA: eight char*, eight ostrstreams

3.1.3 FUNCTIONS:

Calculate_time

Input: char*

Output: none

Return value: clock_t

Description: a string, is passed in and MD5 is applied 1000 on it. The time for all the itterations is calculated and returned. We try to calculate pure MD5 throughput thats why after we finish with MD5 we calculate the time that was spend as loop overhead and we subtract it (We itterate the loop 1000 times without doing anything)

Calculate_throughput

Input: clock_t, float

Output: none

Return value: float

Description: performs only the calculation of the throughput solving the equation.

3.2 **buffercreate**

3.2.1 DESCRIPTION: The file contains only one function that creates strings (buffers) of the requested size.

3.2.2. DATA: none

3.2.3. FUNCTIONS:

Create_buffer

Input: ostrstream &, float

Output: none

Return value: clock_t

Description: an ostrstream object is passed in by reference, and a float. The float is the size in Kbytes of the string that we want the object to contain at the end of the function execution. We output character “a” so many times as the size parameter dictates in the ostrstream object. Then we convert it to a string and we return a reference to this string.

APPENDIX E. MD5 PERFORMANCE TEST CODE

```
#ifndef __buffercreate_H__
#define __buffercreate_H__


//***************************************************************************
//***************************************************************************
// File : buffercreate.cpp
// Author : Ioannis Kondoulis
//
// Operating Enviroment: Windows NT 4.0
// compiler: Borland C++ for Windows, ver. 5.02
// Date: 17 Sep 1998
// Description: This is an auxilliary file that provides the neccesary function
//          that allows the creation of messages in streambuffers for the
//          testing program of MD5. the size of the message that we need to
//          created is passed as a parameter and a reference to a string
//          stream object. Postcondition the strstream object contains a
//          buffer of the sizethat was requested.
//
//
// Assumptions:none
//
// Warnings: none
//
//***************************************************************************
//
#include <fstream>
#include <iostream>
#include <strstream>

ostrstream& create_buffer (ostrstream &sout,  float size);
//void create_file (char* name, float size);

#endif
```

```
//****************************************************************************
//****************************************************************************
// File : buffercreate.cpp
// Author : Ioannis Kondoulis
//
// Operating Enviroment: Windows NT 4.0
// compiler: Borland C++ for Windows, ver. 5.02
// Date: 17 Sep 1998
// Description: This is an auxilliary file that provides the neccesary function
//          that allows the creation of messages in streambuffers for the
//          testing program of MD5
//
//
// Assumptions:none
//
// Warnings: none
//
//****************************************************************************
//
#include "buffercreate.h"

ostrstream& create_buffer (ostrstream &sout,  float size)
{
  int buffersize = static_cast<int>(size * 1024);
  unsigned char fill = 'a';

  for (int i=0 ;i< buffersize;i++)
    sout << fill;

  return sout;
}
```

```
//************************************************************************************
//************************************************************************************
// File : throughputtest.cpp
// Author : Ioannis Kondoulis
//
// Operating Enviroment: Windows NT 4.0
// compiler: Borland C++ for Windows, ver. 5.02
// Date: 17 Sep 1998
// Description: This is the measuring code for the MD5 throughput.The driver
//          creates a series of(eight) messages (in buffers) and calculates
//          the throughput of MD5 for every one of them.
//          The average throughput is also calculated.
//
// Assumptions:none
//
// Warnings: none
//************************************************************************************
//
#include <string.h>
extern "C"{
#include <stdio.h>
#include "hashutils.h"
}
#include <time.h>
#include <dos.h>
#include "buffercreate.h"

clock_t calculate_time (char* inbuffer);
float calculate_throughput(clock_t time, float bufsize);

int main()
{
  float tempthroughput;
  float avgthroughput=0;
```

```
//the messages
char* buffer512 ;
char* buffer1K ;
char* buffer2K ;
char* buffer4K ;
char* buffer8K ;
char* buffer16K ;
char* buffer32K ;
char* buffer64K ;

//create objects that will get the created buffers and will "freeze" them
ostrstream s05out;
ostrstream s1out;
ostrstream s2out;
ostrstream s4out;
ostrstream s8out;
ostrstream s16out;
ostrstream s32out;
ostrstream s64out;

   cout<< "\n The program is creating the messages,"
      << "512 bytes up to 64 Kbytes.";
//fill the buffers up to the requested size and then "freeze" them (to avoid
//any accidental modification of their size. Assign their contents to strings
  buffer512 = create_buffer (s05out,0.5).str();
  cout<<".";
  buffer1K  = create_buffer (s1out,1).str();
  cout<<".";
  buffer2K  = create_buffer (s2out,2).str();
  cout<<".";
  buffer4K  = create_buffer (s4out,4).str();
  cout<<".";
  buffer8K  = create_buffer (s8out,8).str();
  cout<<".";
  buffer16K = create_buffer (s16out,16).str();
```

```
cout<<".";
buffer32K = create_buffer (s32out,32).str();
cout<<".";
buffer64K = create_buffer (s64out,64).str();
cout<<"."<<endl;

tempthroughput = calculate_throughput (calculate_time(buffer512),0.5);
printf ("\n CALCULATED THROUGHPUT FOR BUFFER 512 = %f [Mbits/sec]\n",
                (double)(tempthroughput));
avgthroughput += tempthroughput;

tempthroughput = calculate_throughput (calculate_time(buffer1K),1);
printf ("\n CALCULATED THROUGHPUT FOR BUFFER 1K = %f [Mbits/sec] \n",
                (double)(tempthroughput));
avgthroughput += tempthroughput;

tempthroughput = calculate_throughput (calculate_time(buffer2K),2);
printf ("\n CALCULATED THROUGHPUT FOR BUFFER 2K = %f [Mbits/sec] \n",
                (double)(tempthroughput));
avgthroughput += tempthroughput;

tempthroughput = calculate_throughput (calculate_time(buffer4K),4);
printf ("\n CALCULATED THROUGHPUT FOR BUFFER 4K = %f [Mbits/sec] \n",
                (double)(tempthroughput));
avgthroughput += tempthroughput;

tempthroughput = calculate_throughput (calculate_time (buffer8K),8);
printf ("\n CALCULATED THROUGHPUT FOR BUFFER 8K = %f [Mbits/sec] \n",
                (double)(tempthroughput));
avgthroughput += tempthroughput;

tempthroughput = calculate_throughput (calculate_time(buffer16K),16);
printf ("\n CALCULATED THROUGHPUT FOR BUFFER 16K = %f [Mbits/sec] \n",
                (double)(tempthroughput));
avgthroughput += tempthroughput;
```

```
    tempthroughput = calculate_throughput (calculate_time(buffer32K),32);
    printf ("\n CALCULATED THROUGHPUT FOR BUFFER 32K = %f [Mbits/sec] \n",
                    (double)(tempthroughput));
    avgthroughput += tempthroughput;

   tempthroughput = calculate_throughput (calculate_time(buffer64K),64);
    printf ("\n CALCULATED THROUGHPUT FOR BUFFER 64K = %f [Mbits/sec] \n",
                    (double)(tempthroughput));
    avgthroughput += tempthroughput;

    //calculate the average throughput, divide by the number of messages
    avgthroughput = avgthroughput/8;

    printf ("\n CALCULATED AVERAGE THROUGHPUT  = %f [Mbits/sec] \n",
                                    avgthroughput);
    printf ("\n Press any key to <exit>");
    getchar();
  return (0);
}//end main

//---------------------------------------------------------------------------
//function: calculate_time
//input: char* inbuffer
//output: clock_t
//Remarks: we calculate the MD5 performance on the string that we received
//     as a parameter. We calculate it 1000 times for accuracy purposes.
//---------------------------------------------------------------------------

clock_t calculate_time (char* inbuffer)
{
  clock_t start1_clock;
  clock_t stop1_clock;
  clock_t start2_clock;
  clock_t stop2_clock;
  clock_t return_time;

  //calculate time for 1000 iterations
  start1_clock = clock();
    for(int i = 0 ; i < 1000; i++)
      MD5TestString (inbuffer);
  stop1_clock = clock();

  //calculate the overhead that is irrelevant with MD5
  start2_clock = clock();
    for(int i = 0 ; i < 1000; i++)
  stop2_clock = clock();

  //and subtract it to find the MD performance
  return_time = (stop1_clock-start1_clock)-(stop2_clock-start2_clock) ;

  return return_time;
}

//---------------------------------------------------------------------------
//function: calculate_throughput
//input: clock_t time, float bufsize
//output: clock_t
//Remarks: we calculate the MD5 throughput for a given bufsize.
//      The bufsize is Kbytes so we calculate bits by
//      multiplying by 8*1024.The time that we receive is for 1000 MD5
//      applications on the message and the time is in
//      milliseconds therefore the whole result is Mbits/sec.
//---------------------------------------------------------------------------
float calculate_throughput(clock_t time, float bufsize)
{
  return ((8*bufsize*1024)/(time));
}
//end throughputtest.cpp file
```

APPENDIX F. DOCUMENTATION FOR MAC COMPARISON PERFORMANCE TEST CODE

1. Description

The program measures the comparitive performance of the two MAC comparison functions.

2. Layers

The main uses four functions to calculate the performance:

Calculate_time_same: defines two same strings, applies MD5 on both and calculates how long it takes to execute a comparison of the two message digests one million times. It uses the byte XORing method (it is described in Appendix A).

Calculate2_time_same: defines two same strings, applies MD5 on both and calculates how long it takes to execute a comparison of the two message digests one million times. It uses the integer method (it is described in Appendix A).

Calculate_time_different: defines two different strings, applies MD5 on both and calculates how long it takes to execute a comparison of the two message digests one million times. It uses the byte XORing method (it is described in Appendix A).

Calculate2_time_different: defines two different strings, applies MD5 on both and calculates how long it takes to execute a comparison of the two message digests one million times. It uses the integer method (it is described in Appendix A).

4. Modules

3.1 main

3.1.1 DESCRIPTION: The main just makes the calls to the different functions that execute the comparisons and calculate the execution time and displays the resulst to the user explaining what it does.

3.1.4 DATA: four type clock_t variables (to accommodate the results of the functions)

3.1.5 FUNCTIONS:

Calculate_time_same

Input: none

Output: none

Return value: clock_t

Description: the function declares and defines two *same* strings and calculates their message digests. The 16 byte digests are compared using the byte XORing method (byte by byte XOR ing). The comparison is repeated 1000000 times for averaging the result. The time is calculated and returned.

Calculate2_time_same

Input: none

Output: none

Return value: clock_t

Description: the function declares and defines two *same* strings and calculates their message digests. The 16 byte digests are compared using the integer method (cast to four integers and subtract). The comparison is repeated 1000000 times for averaging the result. The time is calculated and returned.

Calculate_time_different

Input: none

Output: none

Return value: clock_t

Description: the function declares and defines two *different* strings and calculates their message digests. The 16 byte digests are compared using the byte XORing method. The comparison is repeated 1000000 times for averaging the result. The time is calculated and returned.

Calculate2_time_different

Input: none

Output: none

Return value: clock_t

Description: the function declares and defines two *different* strings and calculates their message digests. The 16 byte digests are compared using the integer method. The comparison is repeated 1000000 times for avereging the result. The time is calculated and returned.

APPENDIX G. MAC COMPARISON PERFORMANCE CODE

```
//****************************************************************************
//****************************************************************************
// File : MACompa retest.cpp
// Name : Ioannis Kondoulis
//
// Operating Enviroment: Windows NT 4.0
// compiler: Borland C++ for Windows, ver. 5.02
// Date: 17 Sep 1998
// Description: This is a test program that measures the performance of the two
//          functions that compare 128 bit MACs.The functions themselves are
//          included in the file hashutils.c that interfaces the MD5 code.
//          It uses the clock_t for the measurement of time. Because the time
//          for the comparison is very small we execute it one million times
//          in order to measure it. The result we get is in [msec].
// Assumptions:none
//
// Warnings: none
//
//****************************************************************************
//
#include <string.h>
extern "C"{
#include <stdio.h>
#include "hashutils.h"
}
#include <time.h>
#include <dos.h>

clock_t calculate_time_same ();
clock_t calculate2_time_same ();
clock_t calculate_time_different();
clock_t calculate2_time_different();
```

```
int main()
{
  cout<<  " In the following measurements the two strings are the SAME,"
  <<endl<<"therefore,their message digests are the same.\n"<<endl;

  cout<<"Using the byte XORing method for MAC comparison.\n"
     <<"We execute 10 itterations,results in [nanosecs].\n"<<endl;

  clock_t tempclock;
  for (int i =0; i <10; i++){
    tempclock = calculate_time_same();
    cout<<tempclock<<' ';
  }
  cout<<endl;

  cout<<"\n Using the integer method for MAC comparison.\n"
     <<"We execute 10 itterations,results in [nanosecs].\n"<<endl;
  clock_t tempclock2;
  for (int i =0; i <10; i++){
    tempclock2 = calculate2_time_same();
    cout<<tempclock2<<' ';
  }
  cout<<endl<<endl;

  cout<<  " In the following measurements the two strings are the DIFFERENT,"
  <<endl<<"therefore,their message digests are different.\n"<<endl;

  cout<<"Using the byte XORing method for MAC comparison.\n"
     <<"We execute 10 itterations,results in [nanosecs].\n"<<endl;
  clock_t tempclock3;
  for (int i =0; i <10; i++){
    tempclock3 = calculate_time_different();
    cout<<tempclock3<<' ';
  }
  cout<<endl;
  cout<<"\n Using the integer method for MAC comparison.\n"
     <<"We execute 10 itterations,results in [nanosecs].\n"<<endl;
  clock_t tempclock4;
  for (int i =0; i <10; i++){
    tempclock4 = calculate2_time_different();
    cout<<tempclock4<<' ';
  }
  cout<<endl<<endl;

  cout << "Press any key to <exit>";
  getchar();
  return(0);

}//end main

clock_t calculate_time_same ()
{
  clock_t start1_clock;
  clock_t stop1_clock;
  clock_t start2_clock;
  clock_t stop2_clock;
  clock_t return_time;
  unsigned char* tempchar1;
  unsigned char* tempchar2;

  tempchar1 = MD5String ("Mary had a little lamp");
  tempchar2 = MD5String ("Mary had a little lamp");

  start1_clock = clock();
    for(int i = 0 ; i < 1000000; i++){
      compareDigest(tempchar1,tempchar2);
    }
  stop1_clock = clock();
  start2_clock = clock();
    for(int i = 0 ; i < 1000000; i++){
    //do nothing
    }
  stop2_clock = clock();

  return_time =  (stop1_clock-start1_clock)-(stop2_clock-start2_clock);

  return return_time;
}

clock_t calculate2_time_same ()
{
  clock_t start1_clock;
  clock_t stop1_clock;
  clock_t start2_clock;
  clock_t stop2_clock;
  clock_t return_time;
  unsigned char* tempchar1;
  unsigned char* tempchar2;

  tempchar1 = MD5String ("Mary had a little lamp");
  tempchar2 = MD5String ("Mary had a little lamp");

  start1_clock = clock();

    for(int i = 0 ; i < 1000000; i++){
      compare2Digest(tempchar1,tempchar2);
    }
  stop1_clock = clock();

   start2_clock = clock();
    for(int i = 0 ; i < 1000000; i++){
    //do nothing
    }
   stop2_clock = clock();

   return_time =  (stop1_clock-start1_clock)-(stop2_clock-start2_clock);

   return return_time;
}
clock_t calculate_time_different ()
{
   clock_t start1_clock;
   clock_t stop1_clock;
   clock_t start2_clock;
   clock_t stop2_clock;
   clock_t return_time;
   unsigned char* tempchar1;
   unsigned char* tempchar2;

   tempchar1 = MD5String ("Mary had a little lamp");
   tempchar2 = MD5String ("Mary had a black dog");

   start1_clock = clock();
     for(int i = 0 ; i < 1000000; i++){
       compareDigest(tempchar1,tempchar2);
     }
   stop1_clock = clock();

   start2_clock = clock();
     for(int i = 0 ; i < 1000000; i++){
     //do nothing
     }
   stop2_clock = clock();

   return_time =  (stop1_clock-start1_clock)-(stop2_clock-start2_clock);

   return return_time;
}
```

```
clock_t calculate2_time_different ()
{
  clock_t start1_clock;
  clock_t stop1_clock;
  clock_t start2_clock;
  clock_t stop2_clock;
  clock_t return_time;
  unsigned char* tempchar1;
  unsigned char* tempchar2;

  tempchar1 = MD5String ("Mary had a little lamp");
  tempchar2 = MD5String ("Mary had a black dog");

  start1_clock = clock();

    for(int i = 0 ; i < 1000000; i++){
      compare2Digest(tempchar1,tempchar2);
    }
  stop1_clock = clock();

   start2_clock = clock();
    for(int i = 0 ; i < 1000000; i++){
    //do nothing
    }
  stop2_clock = clock();

  return_time =  (stop1_clock-start1_clock)-(stop2_clock-start2_clock);

  return return_time;
}
//end MAComparetest.cpp file
```

LIST OF REFERENCES

AF-LANE, LAN Emulation over ATM Ver.2 - LUNI Specification, ATM Forum, 1997.

Armitage, J. G., Support for Multicast over UNI3.1-based ATM Networks, Internet Draft, 1995a.

Armitage, J. G., "Multicast and Multiprotocol Support for ATM based Internets." ACM SIGCOMM Computer Communication Review, vol. 25, no. 2, 1995b.

Atkins, D., Puis, P., Hare, C., Kelley, R, Nachenberg, C., Nelson, B. A., Phillips, P., Ritchey, T., Sheldon, T., Snyder, J., Internet Security, Professional Reference, Indianapolis, IN, New Riders, 1997.

Bosselaers, A., Covaerts, R., Vandewalle, J., Fast Hashing on the Pentium, Crypto '96, Springer-Verlag, 1996.

Cheswick, W., Beilovin, S., Firewalls and Internet Security, Reading, MA, Addison Welsey Publishing Company,Inc., 1994.

Davies, D. P., W., Security for Computer Networks, New York, Wiley, 1989.

Galvin, M. J., McCloghrie, K. , Davin, R. J., Secure Management of SNMP Networks, Integrated Network Management II, North Holland, 1991.

Hare, C., Karanjit, S., Internet Firewalls and Network Security, Indianapolis,IN, New Riders Publishing, 1996.

Katz, D., Piscitello, D., Cole, B. Luciani, V., NBMA Next Hop Resolution Protocol (NHRP)., Internet draft, IETF, 1996.

Kercheval, B., TCP/IP over ATM, New Jersey, Prentice-Hall, 1998.

Keshav, S., An Engineering Approach to Computer Networking, Reading, MA, Addixon-Wesley, 1997.

Kohl, J. C. N., B., The Kerberos Network Authentication Service (Ver. 5), RFC 1510, IETF, 1993.

Laubach, M., Classical IP and ARP over ATM, RFC 1577, IETF, 1994.

Merkle, C. R., Secrecy, Authentication and Public Key Systems, Ph.D dissertation, Stanford University, 1979.

Newman, P., Minshall, G., Lyon,T., "IP Switching --ATM Under IP." IEEE/ACM Transactions on Networking, vol. 6, no. 2, pp. 117-128, 1998.

NIST, N. I. o. S. a. T., Digital Signature Standard, NIST FIPS PUB 186, U.S. Department of Commerce, 1994.

Preneel, B., van Oorschot, P., MDx-MAC and Buliding Fast MACs from Hash Functions, Crypto '95, Springer-Verlag LNCS, 1995.

Rekhter, Y. D., B. Katz, D. Rosen, E. Swallow, G., Cisco Systems' Tag Switching Architecture Overview, RFC 2105, N. W. Group, Cisco Systems, Inc., 1997.

Rivest, L. R., The MD4 Message Digest Algorithm, RFC 1186, 1990.

Rivest, L. R., The MD5 Message Digest Algorithm, RFC 1321, 1992.

Schneier, B., Applied Cryptography, New York, Wiley, 1996.

Seaman, M., Smarter and Faster IP Connections, BYTE, pp. 47 - 48, 1997.

Stallings, W., Network and Internetwork Security, New Jersey, Prentice-Hall, Inc, 1995.

Truong, H., Ellington, W., Le Boudec, J., Meier, A., Pace, J., LAN Emulation on an ATM Network, IEEE Communications Magazine, vol. 33, pp. 70-85, 1995.

Tsudik, G., "Message Authetication with One-Way Hash Functions.", ACM Computer Communications Review, vol. 22, no. 5, pp. 29-38, 1992.

Xie, G., Irvine, C., Darroca, G., Kondoulis, I.,"LLPF: An Architecture for Link Layer Packet Filtering", Unpublished manuscript, Department of Computer Science, Naval Postgraduate School, September 1998.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center...2
   8725 John J. Kingman Road, Ste. 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library..2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Dr. Geoffrey Xie...2
   Code CS/Xi
   Naval Postgraduate School
   Monterey, California 93943-5101

4. Dr. Cynthia Irvine..1
   Code CS/Ir
   Naval Postgraduate School
   Monterey, California 93943-5101

5. Dr. G.M. Lundy..1
   Code CS/Ln
   Naval Postgraduate School
   Monterey, California 93943-5101

6. Director, Marine Corps Research Center..1
   MCCDC, Code: C40RC
   2040 Broadway Street
   Quantico, Virginia 22134-5107

7. Don Brutzman, Code UW/Br...1
   Undersea Warfare Department
   Naval Postgraduate School
   Monterey, California 93943-5000

8. Dan Boger..1
   Chairperson, Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943-5000

9. Dr. Blaine Burnham..1
National Security Agency
Research and Development Building
R23
9800 Savage Road
Fort Meade, MD 20755-6000

10. CAPT Dan Galik..1
Space and Naval Warfare Systems Command
PMW 161
Building OT-1, Room 1024
4301 Pacific Highway
San Diego, CA 92110-3127

11. Commander, Naval Security Group Command......................................1
Naval Security Group Headquarters
9800 Savage Road
Suite 6585
Fort Meade, MD 20755-6585
ATTN: Mr. James Shearer

12. Mr. George Bieber..1
Defense Information Systems Agency
Center for Information Systems Security
5113 Leesburg Pike, Suite 400
Falls Church, VA 22041-3230

13. CDR Chris Perry...1
N643
Presidential Tower 1
2511 South Jefferson Davis Highway
Arlington, VA 22202

14. Joseph O'Kane...1
National Security Agency
Research and Development Building
R23
9800 Savage Road
Fort Meade, MD 20755-6000

15. Ioannis Kondoulis...4
36A Velvendous St Kipseli
ATHENS 11363
GREECE